Puncturable Signatures and Applications in Proof-of-Stake Blockchain Protocol

Xinyu Li*, Jing Xu*, Xiong Fan†, Yuchen Wang*, Zhenfeng Zhang*
*Trusted Computing and Information Assurance Laboratory,
Institute of Software, Chinese Academy of Sciences, Beijing, China
†Department of Computer Science, University of Maryland
Email: {xinyu2016, xujing}@iscas.ac.cn, leofanxiong@gmail.com, {wangyuchen, zfzhang}@tca.iscas.ac.cn

***Abstract*—Proof-of-stake (PoS) blockchain protocols are emerging as one of the most promising alternative to the energy-consuming proof-of-work protocols. However, one particularly critical threat in the PoS setting is the well-known long-range attacks caused by secret key leakage (LRSL attack). Specifically, an adversary can attempt to corrupt the secret keys corresponding to accounts possessing substantial stake at some past moment such that double-spend or erase past transactions, violating the fundamental persistence property of blockchain. Puncturable signatures, introduced by Bellare et al. (Eurocrypt 2016), provide a satisfying solution to construct practical proof-of-stake blockchain protocols resilient to LRSL attack, despite of the fact that existent constructions are not efficient enough for practical deployments.**

In this paper, we provide a systematic study of puncturable signatures and explore its applications in proof-of-stake blockchain protocol. The puncturing functionality we desire is for a particular part of message, like prefix, instead of the whole message. We formalize a security model that allows adversary for adaptive signing and puncturing queries, and show a construction with efficient puncturing operation based on Bloom filter data structure and strong Diffie-Hellman assumption. In order to further improve efficiency of puncturing, we introduce another primitive, called tag-based puncturable signature and present a generic construction based on hierarchical identity based signature scheme. Finally, we use puncturable signature to construct practical proof-of-stake blockchain protocols that are resilient to LRSL attack, while previously forward secure signature is used to immunize this attack. We implement our scheme and provide experimental results showing that in comparison with forward secure signatures, our constructions of puncturable signature perform substantially better on signature size, signing and verification efficiency, significantly on key update efficiency.

I. Introduction

Proof-of-stake (PoS) protocols have been heralded as a more ecological way to come to consensus on blockchain since it does not rely on expensive hardware using vast amounts of electricity to compute mathematical puzzles as bitcoin's proof-of-work mechanism. In a proof-of-stake blockchain protocol, roughly speaking, participants randomly elect one party to produce the next block by running a "leader election" process with probability proportional to their current stake (a virtual resource) held on blockchain.

In spite of high efficiency, proof-of-stake blockchains only account for a tiny percentage of existing digital currencies market, mainly due to the fact that most existing proof-of-stake protocols suffer from the well-known long-range attacks (also related to "costless simulation")[41] which degrades security in the blockchain. A oft-cited long-range attack is caused by secret key leakage (abbreviated as LRSL attack in this paper). Specifically, an adversary can attempt to bribe (or corrupt) the secret keys corresponding to accounts that possessed substantial stake at some past moment (however, currently low-stake), and then construct a fork and alter the history from the point in the past when he controls majority. In this case, the adversary can continue to hold majority stake (e.g., by the reward fees of generating or issuing blocks) such that the attack can be sustained.

Puncturable signature (PS), introduced by Bellare et al. [12], provides a satisfying solution to construct practical proof-of-stake blockchain protocols resilient to LRSL attack. Loosely speaking, a puncturable signature scheme provides a Puncture functionality that, given a secret key and a message m, produces an updated secret key that is able to sign all messages except for the punctured message m. In this paper, we further generalize the definition of puncturable signature, particularly, the strings associated with the punctured signing key can be any part of signed messages (e.g. its prefix). In proof-of-stake protocols, the leader U (elected for issuing block) signs the block B_i with puncturable signature by secret key sk_U at some time slot sl_i, where sl_i is the part of block B_i, and then U performs puncturing operation on message sl_i which results in an updated sk'_U. More specifically, if no empty block is allowed, the punctured message can be $H(B_{i-1})$ instead of sl_i, where $H(B_{i-1})$ is the part of block B_i and B_{i-1} is the previous block of B_i. The security of puncturable signature guarantees that anyone cannot sign another data block B'_i with the same sl_i (or $H(B_{i-1})$) even though sk'_U is exposed, and thus LRSL attack can be avoided.

A natural way to remedy LRSL attack in proof-of-stake blockchain protocols is to use forward secure signature [11], which preserves the validity of past signatures even if the current secret key is compromised. However, the computation performance of forward secure signature depends on either the time periods set in advance logarithmically (even linearly) or the time periods elapsed so far, which brings undesirable consumption and becomes a fatal issue for blockchain applications. Moreover, most signers have no chance to do any signing within one period but they have to update the signing key as long as the current period ends, which makes the update operation a vain effort in the proof-of-stake blockchain. In fact, forward secure signature can be treated as one special kind of puncturable signature where the punctured message is earlier period of time.

Puncturable signatures can also be used in many other scenarios such as asynchronous transaction data signing services. Transaction data signing is a process which guarantees the integrity and authenticity of the sensitive transaction data, such as payment instruction or transaction information of buying a real estate offering. In many cases, using ordinary digital signatures is not enough for these application, as they often fail to ensure the integrity of past messages in the case when a user's key is compromised. This is particularly challenging in non-interactive and asynchronous message system, where users may not be online simultaneously and messages may be delayed for substantial periods due to delivery failures and connectivity issues. Similar problem also exists in theoretical part. For instance, in non-interactive multiparty computation (NI-MPC) [31], where a group of completely asynchronous parties can evaluate a function (e.g. for the purpose of voting) over their joint inputs by sending a signed message to an evaluator who computes the output. The adversary would control the final output if he can corrupt some parties within a period of time. In these examples, the transaction session ID can be used as a prefix, and after the honest user signs the transaction data (or message), the prefix is punctured so that no other signature exists for messages agreeing on the same prefix. Therefore, the integrity of transaction data (or message) is ensured by puncturable signatures.

A. Our Contributions

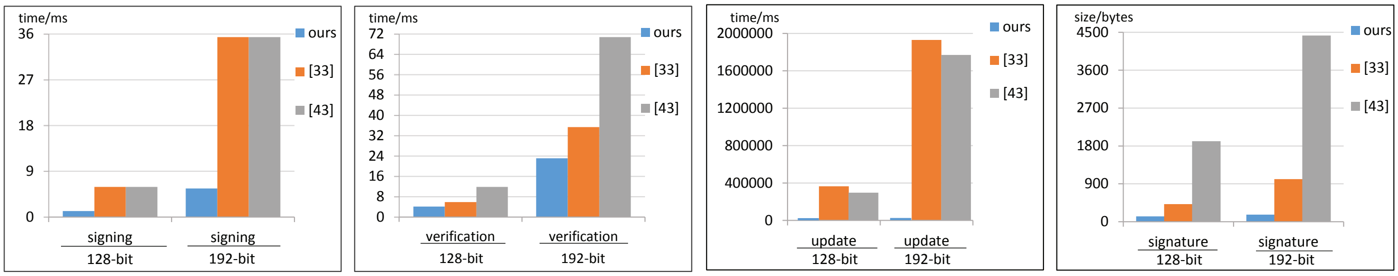
In this work, we provide a systematic study of puncturable signature and its applications in proof-of-stake blockchain protocol. Our overall goal is to design puncturable signature that allows for fine-grained revocation of signing capability with minimum computation cost, and make it a suitable building block to construct secure and practical proof-of-stake blockchain protocol. More specifically, our technical contributions are threefold.

Puncturable signature and its construction. We introduce the notion of puncturable signature with extended puncturing functionality where the secret key can be updated by puncturing any particular part of message (for simplicity, we use the prefix of message in this paper) instead of puncturing the whole message. In the security model we propose, in addition to making adaptive signing and puncturing queries, adversary also has (one-time) oracle access to a featured Corruption oracle, by which the adversary can obtain the current secret key if the challenging string is in the puncturing set P. Then we show a construction of puncturable signature based on the probabilistic Bloom filter data structure [13] that is secure under our security model. Our PS construction is inspired by an elegant work [24], where the authors show how to construct puncturable encryption based on Bloom filter. However, different from the expanded (k times) ciphertext size of underlying encryption scheme in [24], in our construction, the signature size is almost equal to that of the underlying signature scheme.

In comparison with two prior puncturable signature schemes [12][31], our construction achieves significant efficiency improvement in both signing and puncturing operations. More specifically, the construction in [12] relies on indistinguishability obfuscation, which incurs prohibitive computational burden in practice, while the other one [31] needs update public key for every puncturing, which has some theoretical merits but hard to implement in real world deployment. On the contrary, in our construction, a puncturing operation only involves a small number of efficient computations (i.e. hashing), plus the deletion of certain parts of the secret key, which outperforms previous schemes by orders of magnitude. Indeed, puncturable signature is not a simple inverse operation of puncturable encryption, which is also the reason for no efficient puncturable signature scheme even though efficient puncturable encryption constructions have been proposed for a long time. The crucial difficulty in designing puncturable signature scheme is how to bind the private key with punctured messages such that the updated private key cannot sign for punctured messages.

Tag-based puncturable signature. For our puncturable signature scheme based on Bloom filter, the signing algorithm may output $\perp$ for messages whose prefix is not punctured. This is caused by the false positive probability in Bloom filter, and the probability it happens is closely related to the size of secret key and the number of puncturing performed. Put simply, the lower the error probability, the larger the size of secret key and the smaller number of puncturing performed. Therefore, to maintain a balance between space efficiency and error probability, we introduce a new primitive, called tag-based puncturable signatures. In particular, in the lifetime of public key, an ordered tag is updated as long as puncturing operation times reach a pre-set limit, and correspondingly the Bloom filter is reset. We present a generic construction based on Bloom filter from hierarchical identity based signature (HIBS) scheme, and prove our construction is secure against adaptive puncturing if the underlying HIBS is secure. The intuition behind the construction combines a binary tree approach with our construction of puncturable signatures, where each tag corresponds to a leaf of an ordered binary tree of depth d. Our tag-based construction and its security analysis are independent of any particular instantiation of building blocks, HIBS and PS.

Applications in Proof-of-stake blockchain protocol. We use puncturable signature to construct practical proof-of-stake blockchain protocols that are resilient to LRSL attack. Ouroboros Paros [23], a proof-of-stake blockchain protocol, has a real-world implementation in Cardano platform. We present an ideal functionality $\mathcal{F}_{\mathsf{PS}}$ of puncturable signature scheme, and replace $\mathcal{F}_{\mathsf{KES}}$ of forward secure signature schemes in Ouroboros Paros [23] protocol with $\mathcal{F}_{\mathsf{PS}}$. Then we show that the properties (common prefix, chain quality and chain growth) of Ouroboros Paros protocol remain true in the replaced setting. However, most of the existing forward secure signatures have poor performance on key update as well as other operations, often depending on the time period number linearly, which is unsuitable for blockchain application. We conduct experiments evaluating the overhead of deploying our puncturable signature construction and existing forward security signature schemes [33][43] at both 128-bit and 192-bit security levels. Figure 1 illustrates the efficiency comparison and the results show that our scheme performs substantially better on signature size, signing and verification efficiency, significantly on key update efficiency, which reduces both communication and computation complexity. In fact, we can replace the ordinary signature with our puncturable signature construction in any other proof-of-stake protocols such as Ouroboros [36] and Snow White [22] protocols. Due to the



(a) Sign time (b) Verify time (c) Update time (d) Signature size

Figure 1: Efficiency Comparison

fact that our construction can retain the efficiency of the underlying scheme on signing and verifying, with additional k hash functions, the improved protocols can provide resilience to LRSL attack at almost no additional computing cost.

B. Related Work

Puncturable signature. A puncturable signature scheme allows to update its signing key sk for an arbitrary message m such that the resulting punctured signing key can produce signatures for all messages except for m. It is introduced by Bellare et al. [12] as a tool to prove a negative results on differing-inputs obfuscation. However, their construction is based on indistinguishability obfuscation [26] and one-way function, thus, do not yield practical schemes. Moreover, it requires that the punctured signing key is associated with the full signed message. In contrast our construction is based on τ-SDH assumption and the associated messages with the punctured signing key can be any particular part of signed messages (e.g. the prefix of message), which is more flexible and applicable. Halevi et al. [31] also propose a puncturable signature scheme which is puncturable at any prefix of the signed message. However, their puncturable operation needs to update public keys repeatedly. In practice, it is inefficient to verify the updated public keys continuously and it is also difficult to let each user in the system maintain other users' public keys updated.

Delegatable Signature. Policy-based signature, introduced by Bellare et al. [10], allows a signer to only sign messages conforming to some authority-specified policy. It elegantly unifies existing work, capturing other forms of signatures as special cases. Puncturable signature differs from this work, as secret key is updated adaptively with respect to message prefix. Another related primitive, called functional signature is introduced in work [15]. In functional signature, in addition to a master signing key that can be used to sign any message, there are signing keys for a function f, which allow one to sign any message in the range of f. Delegatable functional signature is introduced in work [6] and supports the delegation of signing capabilities to another party, called the evaluator, with respect to a functionality. Append-only signatures (AOS) [37] is also a related primitive, in particular, any party given an AOS signature on a message $m = M_1||...||M_n$ can compute an AOS signature on any message $m = M_1||...||M_n||M_{n+1}$. Different from above primitives, puncturable signature provides a puncture functionality that may repeatedly update the secret key to revoke signing capability for selected messages besides providing delegation function.

Forward secure signature. A forward secure signature scheme guarantees the adversary with the compromised secret key at some point in time cannot forge signatures relative to previous time periods. It is introduced by Anderson [5] and formalized by Bellare et al. [11]. The constructions of prior forward secure signatures are divided into two categories: using arbitrary signature schemes in a black box manner [39][43], and modifying specific signature schemes [4][11][33]. All these forward secure schemes except for [43], the number of time periods T (arbitrarily large) must be set in advance, such that the performance depends on T logarithmically or even linearly. Nevertheless, the performance in [43] still depends on the time periods elapsed so far.

Proof-of-stake blockchain protocol. Proof-of-stake protocols were first initiated in online forums and subsequently a number of proof-of-stake protocols were proposed and implemented by the academic community. In order to provide forward security (and also achieve resilience against LRSL attack and other long-range attack), Ouroboros Paros [23] and Ouroboros Genesis [7] formalize and realize in the universal composition setting a forward secure digital signature scheme, Algorand [28] considers it as one of future work and implements ephemeral key pairs in its updated full version [20], whereas Snow White [22] and Ouroboros [36] adopt a weak adaptive corruption model and cannot avoid LRSL attack. In addition, several countermeasures have been proposed, such as punishment mechanism revealing the real identity [41] or the signing key [25] of the malicious stakeholder, the trusted execution environments [41], and checkpointing mechanism [22].

II. PRELIMINARIES

Notation. Let λ denote the security parameter, $\lfloor x \rfloor$ denote the greatest integer less than or equal to x, $[n]$ denote the set of the first n positive integers, and PPT denote probabilistic polynomial time. For an array $T \in \{0,1\}^n$, we let $T[i]$ denote the i-th bit of the array, if $i \leq n$.

We say a function $\mathsf{negl}(\cdot) : \mathbb{N} \to (0,1)$ is negligible, if for every constant $c \in \mathbb{N}$, $\mathsf{negl}(n) < n^{-c}$ for sufficiently large n.

A. Bloom Filter

A Bloom filter [13] is a probabilistic data structure for the approximate set membership problem. It allows a succinct representation T for set $\mathcal{S}$ of elements from a large universe $\mathcal{U}$. Put simply, a query to Bloom filter always outputs 1 ("yes") for element $s \in \mathcal{S}$, and ideally it always outputs 0 ("no") for element $s \notin \mathcal{S}$. However, the succinctness of Bloom filter comes at the cost that for any query $s \notin \mathcal{S}$ the answer can also be 1, with small probability (called the *false-positive probability*).

Definition 1 (Bloom Filter)**.** *A Bloom filter* BF *for set* $\mathcal{U}$ *consists of algorithms* $(\mathsf{Gen}, \mathsf{Update}, \mathsf{Check})$*, which are defined as follows.*

- $\mathsf{Gen}(\ell, k)$: On input two integers $\ell, k \in \mathbb{N}$, the algorithm first samples k universal hash functions $H_1, \dots, H_k$, where $H_j : \mathcal{U} \to [\ell]$ $(j \in [k])$. Set $H = \{H_i\}_{i\in[k]}$ and $T = 0^\ell$ (T is an ℓ-bit array with all bits set to 0). Output (H, T).
- $\mathsf{Update}(H, T, u)$: On input $H = \{H_i\}_{i\in[k]}$, $T \in \{0,1\}^\ell$, and $u \in \mathcal{U}$, the algorithm defines the updated state T' by first assigning $T' = T$. Then, it sets $T'[H_i(u)] = 1$ for all $i \in [k]$, and returns T'.
- $\mathsf{Check}(H, T, u)$: On input $H = \{H_i\}_{i\in[k]}, T \in \{0,1\}^\ell$, and $u \in \mathcal{U}$, the algorithm returns a bit $b = \wedge_{i\in[k]} T[H_i(u)]$.

Properties of Bloom filter. The properties of Bloom filter relevant to our work can be summarized as follows:

Perfect completeness: A Bloom filter always outputs 1 for elements that have already been added to the set $\mathcal{S}$. More precisely, let $\mathcal{S} = \{s_1, \dots, s_n\} \in \mathcal{U}^n$ be any vector of n elements of $\mathcal{U}$. Let $(H, T_0) \leftarrow \mathsf{Gen}(\ell, k)$ and for $i \in [n]$, set $T_i = \mathsf{Update}(H, T_{i-1}, s_i)$. Then for any $s^* \in \mathcal{S}$ and any $(H, T_0) {\leftarrow} \mathsf{Gen}(\ell, k)$, we have $\mathbf{Pr}[\mathsf{Check}(H, T_n, s^*) = 1] = 1$.

Compact representation of set $\mathcal{S}$: The size of representation T is a constant number of ℓ bits, and independent of the size of set $\mathcal{S}$ and the representation of individual elements of $\mathcal{U}$. The increase in size of set $\mathcal{S}$ only increases the *false-positive probability*, but not the size of representation T.

Bounded false-positive probability: Given the size of set $\mathcal{S}$, the probability that an element which has not yet been added to the Bloom filter is erroneously "recognized" as being in the filter can be made arbitrarily small, by choosing ℓ and k accordingly. More precisely, let $\mathcal{S} = \{s_1, \dots, s_n\} \in \mathcal{U}^n$ be any vector of n elements of $\mathcal{U}$, for any $s^* \in \mathcal{U} - \mathcal{S}$, we have $\mathbf{Pr}[\mathsf{Check}(H, T_n, s^*) = 1] \approx (1 - e^{kn/\ell})^k$, where $(H, T_0) \leftarrow \mathsf{Gen}(\ell, k)$, $T_i = \mathsf{Update}(H,\ T_{i-1},\ s_i)$ for $i \in [n]$, and the probability is taken over the random coins of algorithm $\mathsf{Gen}(\ell, k)$.

Discussion on the choice of parameters. In bloom filter, assuming the optimal number of hash function k to achieve the smallest false-positive probability pr, we obtain a size of the bloom filter given by $\ell = -\frac{n \ln pr}{(\ln 2)^2}$, and the optimal k is given by $k = \lceil \frac{\ell}{n} \ln 2 \rceil$. Recall the instance in [24], when $pr = 10^{-3}$, we have $\ell \approx 2$ MB and $k = 10$.

B. Bilinear Groups

We say that $\mathcal{G}$ is a bilinear group generator if given the security parameter λ, it outputs a tuple $\mathsf{params} = (p, e, \psi, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T,\ P_1, P_2)$, where $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are three groups have prime order p, P_i is the generator of $\mathbb{G}_i$ for $i \in \{1, 2\}$, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerate map satisfying:

- Bilinearity: For any $(P, Q) \in \mathbb{G}_1 \times \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p^*$, we have $e(aP, bQ) = e(P, Q)^{ab}$.
- Non-degeneracy: For any $P \in \mathbb{G}_1$, $e(P, Q) = 1$ for any $Q \in \mathbb{G}_2$ iff $S = \mathcal{O}$.
- Computability: There is an efficient algorithm to compute $e(P, Q)$ for any $(P, Q) \in \mathbb{G}_1 \times \mathbb{G}_2$.
- There exists an efficiently, publicly computable isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$ such that $\psi(Q) = P$.

The security of our scheme is based on the τ-strong Diffie-Hellman (τ-SDH) assumption, which was previously formalized in [14] and [44].

Definition 2 (τ-Strong Diffie-Hellman Assumption (τ-SDH))**.** *Let* $\mathsf{params} = (p, e, \psi, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, P_1, P_2) \leftarrow \mathcal{G}(1^\lambda)$*, and let* $(P_1,\ P_2,\ \alpha P_2, \alpha^2 P_2, \dots, \alpha^\tau P_2)$ *be a* $\tau + 2$ *tuple for* $\alpha \in \mathbb{Z}_p^*$*. We say* τ*-SDH assumption holds if for any* PPT *adversary* $\mathcal{A}$*,* $\mathbf{Pr}[(h, \frac{1}{h+\alpha} P_1) \leftarrow \mathcal{A}(P_1, P_2, \alpha P_2, \alpha^2 P_2, \dots, \alpha^\tau P_2)] \leq \mathsf{negl}(\lambda)$ *with* $h \in \mathbb{Z}_p^*$*.*

C. Hierarchical Identity-Based Signature

We recall the syntax and security definition of hierarchical identity-based signature (HIBS) [21][27].

Definition 3 (Hierarchical Identity-based Signature(HIBS))**.** *A* t*-level hierarchical identity-based signature scheme with identity space* $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_t$ *consists of the following algorithms:*

- $\mathsf{Setup}(1^\lambda)$: On input a security parameter λ, the algorithm outputs the master public key mpk and the root secret key sk_ε.
- $\mathsf{Delegate}(\mathsf{sk}_\tau, d)$: On input the secret key sk_τ ($\tau \in \mathcal{D}_1 \times \cdots \times \mathcal{D}_{i-1}$) and $d \in \mathcal{D}_i$, the algorithm outputs a secret key $\mathsf{sk}_{\tau|d}$.
- $\mathsf{Sign}(\mathsf{sk}_\tau, m)$: On input the secret key sk_τ and a message m, the signing algorithm outputs a signature σ.
- $\mathsf{Verify}(\mathsf{mpk}, \tau, m, \sigma)$: On input the identity τ, a signature σ and message m, the verification algorithm outputs 1 if σ is a valid signature of message m signed by τ. Otherwise, it outputs 0.

Definition 4 (Correctness)**.** *For any message* m *and any* $(\mathsf{mpk}, \mathsf{sk}_\varepsilon) \leftarrow \mathsf{Setup}(1^\lambda)$*, we have* $\mathsf{Verify}(\mathsf{mpk}, \tau, m, \mathsf{Sign}(\mathsf{sk}_\tau, m))$ = *1.*

Security Definition. For the security definition of HIBS, we use the following experiment to describe it. Formally, for any PPT adversary $\mathcal{A}$, we consider the experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{hibs}}(1^\lambda)$ between adversary $\mathcal{A}$ and challenger $\mathcal{C}$:

1) **Setup**: $\mathcal{C}$ computes $(\mathsf{mpk}, \mathsf{sk}_\varepsilon) \leftarrow \mathsf{Setup}(1^\lambda)$ and sends mpk to adversary $\mathcal{A}$. $\mathcal{C}$ also initializes two empty sets Q_{sign} and Q_{key}.
2) **Queries**: Proceeding adaptively, adversary $\mathcal{A}$ can submit the following two kinds of queries:
   - Signing queries: On input identity τ and message m from adversary $\mathcal{A}$, $\mathcal{C}$ computes $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}_\tau, m)$ and sends back σ. $\mathcal{C}$ also puts (τ, m) into set Q_{sign}.
   - Key queries: On input identity τ from adversary $\mathcal{A}$, $\mathcal{C}$ returns a secret key sk_τ by computing $\mathsf{Delegate}(\mathsf{sk}_\epsilon, \tau)$. $\mathcal{C}$ also puts τ into set Q_{key}.
3) **Forgery**: Adversary $\mathcal{A}$ outputs a forgery (τ^*, m^*, σ).

We say that the forgery wins experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{hibs}}(1^\lambda)$ if there does not exist $\tau \in Q_{\mathsf{key}}$, such that τ is τ^* or prefix of τ^*, and

$$(\tau^*, m^*) \notin Q_{\mathsf{sign}} \wedge \mathsf{Verify}(\mathsf{mpk}, \tau^*, m^*, \sigma) = 1$$

Definition 5. *We say the HIBS scheme is unforgeable, if for any* PPT *adversary $\mathcal{A}$, the probability of winning experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{hibs}}(1^\lambda)$ is $\mathsf{negl}(\lambda)$, where the probability is over the randomness of the challenger and adversary.*

III. Puncturable Signatures

In this section, we formalize the syntax and security definition of puncturable signature, and then we propose a puncturable signature scheme and prove its security under τ-SDH assumption.

A. Syntax and Security Definition

Let the message space be $\mathcal{M}$. A puncturable signature scheme Σ consists of a tuple of PPT algorithms $\Sigma = (\mathsf{Setup}, \mathsf{Puncture}, \mathsf{Sign}, \mathsf{Verify})$ with descriptions as follows:

- $\mathsf{Setup}(1^\lambda, \ell, k)$: On input the security parameter λ, parameters ℓ and k for the Bloom filter, the setup algorithm outputs public key vk, secret key sk.
- $\mathsf{Puncture}(\mathsf{sk}, str)$: On input the secret key sk and a string $str \in \mathcal{M}$, the puncturing algorithm outputs updated secret key sk'. We also say that str has been punctured.
- $\mathsf{Sign}(\mathsf{sk}, m)$: On input the secret key sk and a message m, it outputs a signature σ.
- $\mathsf{Verify}(\mathsf{vk}, m, \sigma)$: On input the public key vk, a signature σ and message m, the verification algorithm outputs 1 if σ is a valid signature for m. Otherwise, it outputs 0.

Correctness of puncturable signature. Intuitively, the correctness requires that (1) signing is always successful with the initial, non-punctured secret key, (2) signing fails when attempting to sign a message with a prefix that has been punctured, and (3) the probability that signing fails is bounded by some non-negligible function, if the prefix of the message to be signed has not been punctured.

Definition 6 (Correctness)**.** *For any message m with prefix m', any $(\mathsf{sk}_{\mathsf{init}}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda)$, and any sequence of invocations of $\mathsf{sk} \leftarrow \mathsf{Puncture}(\mathsf{sk}, .)$, we have*

1) $\mathsf{Verify}(\mathsf{vk}, m, \mathsf{Sign}(\mathsf{sk}_{\mathsf{init}}, m)) = 1$, *where $\mathsf{sk}_{\mathsf{init}}$ is the initial, non-punctured secret key.*

2) If m' has been punctured, then $\mathsf{Verify}(\mathsf{vk}, m, \mathsf{Sign}(\mathsf{sk}', m)) = 0$.

3) Otherwise, it holds that $\Pr[\mathsf{Verify}(\mathsf{vk}, m, \mathsf{Sign}(\mathsf{sk}, m)) \neq 1] \leq \mu(\ell, k)$, *where $\mu(\cdot)$ is some (possibly non-negligible) bound.*

Remark 1. We note that the puncturing functionality defined above is for message-prefix, whose length can be determined in specific implementation (e.g., the slot parameter in proof-of-stake blockchain). In specific applications, the message m to be signed can be split into n ($n \geq 1$) parts denoted by $m = m_1||...||m_i||...||m_n$, where different parts may have different lengths and different semantics, for example, m_1 denotes the time stamp and the remaining denotes the message specifics. We can extend the puncturing functionality by puncturing strings at arbitrarily pre-defined position (even the whole message), e.g. i-th part, which means the signing algorithm fails for message $m = m_1\|...\|m_i\|...\|m_n$ if m_i has been punctured. For simplicity, we still use prefix-puncturing through this paper, but both the definitions and our constructions can be easily extended to support the extended puncturing functionality.

Security Definition. For the security definition of puncturable signature Σ, we use the following experiment to describe it. Formally, for any PPT adversary $\mathcal{A}$, we consider the experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{ps}}(1^\lambda)$ between $\mathcal{A}$ and challenger $\mathcal{C}$:

1) **Setup**: $\mathcal{C}$ computes $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$ and sends vk to adversary $\mathcal{A}$. The $\mathcal{C}$ initializes two empty sets $Q_{\mathsf{sig}} = \emptyset$ and $P = \emptyset$.
2) **Query Phase**: Proceeding adaptively, adversary $\mathcal{A}$ can submit the following two kinds of queries:
   - **Signature query**: On input message m from adversary $\mathcal{A}$, $\mathcal{C}$ computes $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ and updates $Q_{\mathsf{sig}} = Q_{\mathsf{sig}} \cup \{m\}$. Then $\mathcal{C}$ sends back σ.
   - **Puncture query**: On input a string str, $\mathcal{C}$ updates sk by running $\mathsf{Puncture}(\mathsf{sk}, str)$, and updates P=$P \cup \{str\}$.
3) **Challenge Phase**: $\mathcal{A}$ sends the challenge puncture string m' to challenger, and $\mathcal{A}$ can still submit signature and puncture queries as described in **Query phase**.
4) **Corruption query**: The challenger returns sk if $m' \in P$ and $\perp$ otherwise.
5) **Forgery**: $\mathcal{A}$ outputs a forgery pair (m, σ).

We say that adversary $\mathcal{A}$ wins the experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{ps}}(1^\lambda)$ if m' is the prefix of m and $m \notin Q_{\mathsf{sig}} \wedge \mathsf{Verify}(\mathsf{vk}, m, \sigma) = 1$.

Definition 7 (Unforgeability with adaptive puncturing)**.** *We say the puncturable signature scheme Σ is unforgeable with adaptive puncturing, if for any* PPT *adversary $\mathcal{A}$, the probability of winning experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{ps}}(1^\lambda)$ is $\mathsf{negl}(\lambda)$, where the probability is over the randomness of the challenger and adversary.*

B. Our Construction

We present a puncturable signature construction based on Chinese IBS, an identity-based signature scheme standardized in ISO/IEC 14888-3 [32].

The key idea of our construction is to derive secret keys for all Bloom filter bits $i \in [l]$ using IBS schemes, and then bind the prefix string m' with k positions where the secret keys are used to sign messages with prefix m'. In addition, puncturing at m' implies the deletion of keys in corresponding positions.

Let $(p, e, \psi, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, P_1, P_2) \leftarrow \mathcal{G}(1^\lambda)$, and $\mathsf{BF} = (\mathsf{BF.Gen}, \mathsf{BF.Update}, \mathsf{BF.Check})$ be a Bloom filter. Choose a random generator $P_2 \in \mathbb{G}_2$, and set $P_1 = \psi(P_2) \in \mathbb{G}_1$. Let $h_1 : \mathbb{N} \to \mathbb{Z}_p^*$ and $h_2 : \{0,1\}^* \times \mathbb{G}_T \to \mathbb{Z}_p^*$ be cryptographic hash functions, which we model as random oracles in the security proof. The public parameters are $\mathsf{params} := (p, e, \psi, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, P_1, P_2, h_1, h_2)$ and all the algorithms described below implicitly take params as input. The construction of puncturable signature scheme $\mathsf{ps} = (\mathsf{Setup}, \mathsf{Puncture}, \mathsf{Sign}, \mathsf{Verify})$ is the following:

- $\mathsf{Setup}(1^\lambda, \ell, k)$: This algorithm first generates a Bloom filter instance by running $(\{H_j\}_{j\in[k]}, T) \leftarrow \mathsf{BF.Gen}(\ell, k)$. Then it chooses $s \xleftarrow{\$} \mathbb{Z}_p^*$ and outputs
$$\mathsf{sk} = (T, \{sk_i\}_{i\in[\ell]}), \quad \mathsf{vk} = (P_{pub}, g, \{H_j\}_{j\in[k]}),$$
where $sk_i = \frac{s}{s+h_1(i)}P_1$, $P_{pub} = sP_2$, and $g = e(P_1, P_{pub})$.
- $\mathsf{Puncture}(\mathsf{sk}, str)$: Given a secret key $\mathsf{sk} = (T, \{\mathsf{sk}_i\}_{i\in[\ell]})$ where $\mathsf{sk}_i = \frac{s}{s+h_1(i)}P_1$ or $\perp$, this algorithm first computes T' = $\mathsf{BF.Update}$ $(\{H_j\}_{j\in[k]}, T, str)$. Then, for each $i \in [\ell]$ define
$$\mathsf{sk}'_i = \begin{cases} \mathsf{sk}_i, & \text{if } T'[i] = 0 \\ \perp, & \text{otherwise} \end{cases}$$
where $T'[i]$ denotes the i-th bit of T'. Finally, the algorithm returns $\mathsf{sk}' = (T', \{\mathsf{sk}'_i\}_{i\in[\ell]})$.
- $\mathsf{Sign}(\mathsf{sk}, m)$: Given a secret key $\mathsf{sk} = (T, \{\mathsf{sk}_i\}_{i\in[\ell]})$ and a message m with prefix m', the algorithm first checks whether $\mathsf{BF.Check}$ $(\{H_j\}_{j\in[k]}, T, m') = 1$ and outputs $\perp$ in this case. Otherwise, note that $\mathsf{BF.Check}(\{H_j\}_{j\in[k]}, T, m')$ = 0 means there exists at least one index $i_j \in \{i_1, \cdots, i_k\}$ such that $\mathsf{sk}_{i_j} \neq \perp$, where $i_j = H_j(m')$ for $j \in [k]$. Then it picks a random index i_{j^*} such that $\mathsf{sk}_{i_{j^*}} = \frac{s}{s+h_1(i_{j^*})}P_1$. Choose $x \xleftarrow{\$} \mathbb{Z}_p^*$ and compute $r = g^x$, then set
$$h = h_2(m, r), S = (x - h)\mathsf{sk}_{i_{j^*}}.$$
The output signature on m is $\sigma = (h, S, i_{j^*})$.
- $\mathsf{Verify}(\mathsf{vk}, m, \sigma)$: Given public key $\mathsf{vk} = (P_{pub}, g, \{H_j\}_{j\in[k]})$, a message m with prefix m', and a signature $\sigma = (h, S, i_{j^*})$, the algorithm checks whether
$$i_{j^*} \in S_{m'} \wedge h = h_2\big(m, e(S, h_1(i_{j^*})P_2 + P_{pub})g^h\big),$$
where $S_{m'} = \{H_j(m') : j \in [k]\}$. If it holds, the algorithm outputs 1, and 0 otherwise.

Lemma 1. *Our basic construction described above satisfies correctness (c.f. Definition 6).*

Proof: If the secret key is initial and non-punctured, we have
$$\begin{aligned} & e(S, h_1(i_{j^*})P_2 + P_{pub})g^h \\ & = e((x-h).\frac{s}{h_1(i_{j^*}) + s}P_1, (h_1(i_{j^*}) + s)P_2).e(P_1, P_2)^{hs} \\ & = e(P_1, P_2)^{xs-hs}e(P_1, P_2)^{hs} = g^x = r \end{aligned}$$
and then $h = h_2\big(m, r\big) = h_2(m, e(S, h_1(i_{j^*})P_2 + P_{pub})g^h\big)$. Therefore, the first requirement of Definition 6 holds. If m' is punctured, by the perfect completeness of Bloom filter, we have $\mathsf{BF.Check}(H, T, m')$ = 1. Therefore, the signing of the message m with prefix m' fails and the second requirement of Definition 6 holds. If m' is not punctured, the correctness error of our construction occurs only when $\mathsf{BF.Check}(H, T, m') =$ 1, which is essentially identical to the false-positive probability of the Bloom filter and the third requirement of Definition 6 holds. ■

Remark 2. In this section, we assume that the false-positive probability from Bloom filter is acceptable, which means the number of puncturings supported by our construction is below a pre-set parameter, depending on Bloom filter and the application scenarios. In the security proof below, we also assume that the number of puncturing queries is also bounded by the pre-set parameter. We will discuss how to handle a larger number of punturings in Section IV.

Theorem 1. *Assuming that an algorithm $\mathcal{A}$ wins in the $\mathbf{Expt}^{\mathsf{ps}}_{\mathcal{A}}(1^\lambda)$ experiment (c.f. Definition 7) to our construction* ps*, with advantage $\epsilon_0 \geq 10k(q_S + 1)(q_S + q_{h_2})/\big(p(1 - (1 - 1/l)^k)\big)$ within running time t_0, τ-SDH assumption can be broken for $\tau = q_{h_1}$ within running time $t_2 \leq 120686 q_{h_2} t_0/(\epsilon_0(1 - \tau/p))$, where q_{h_1}, q_{h_2} and q_S are maximum query times of hash function h_1, h_2 and signing respectively.*

Due to space limit, we present the full proof in Appendix A. Our proof is split into two steps. First, if there exists an adversary $\mathcal{A}$ wins in the $\mathbf{Expt}^{\mathsf{ps}}_{\mathcal{A}}(1^\lambda)$, we can construct another adversary $\mathcal{B}$ that wins in a weaker attack experiment denoted by $\mathbf{Expt}^{\mathsf{fps}}_{\mathcal{A}}(1^\lambda)$, where $\mathcal{B}$ is challenged on a fixed position by the challenger and $\mathcal{B}$ has to output a forgery on messages at that position. Second, we prove the τ-SDH assumption would be broken if such an adversary $\mathcal{B}$ exists according to the forking lemma [47].

IV. Tag-Based Puncturable Signatures

For a puncturable signature scheme based on Bloom filter, the false-positive probability would exceed an acceptable bound if the secret key has been punctured for many times, and thus the signing key pair has to be updated for an empty bloom filter. In this part, we formalize and construct a new primitive, tag-based puncturable signatures, where the lifetime of the public key is split into several periods, during each period secret key can be punctured for at most max times[1]. We use a tag τ (initialized to 1) to identify the valid period of the signing key, and τ increases by one for next period. The main advantage of this approach is that for a given bound of correctness error, we can handle the number of puncturing per valid period as in the basic scheme during the entire life time of the public key. This is inspired by the time-based approach used to construct puncturable encryption [24][29][30], which in turn is inspired by the construction of forward-secure PKE by Canetti et al. [18].

A. Syntax and Security Definition

A tag-based puncturable signature scheme Σ consists of a tuple of PPT algorithms Σ = (Setup, PuncStr, PuncTag, Sign, Verify) with descriptions as follows:

- $\mathsf{Setup}(1^\lambda, \ell, k, t)$: On input the security parameter λ, parameters ℓ and k for the Bloom filter, and t specifying the number of time periods, the setup computes the secret/public keys $(\mathsf{sk}, \mathsf{vk})$, and initialize a tag τ_0. It then outputs public key vk, secret key sk associated to tag τ_0.
- $\mathsf{PuncStr}(\mathsf{sk}, str)$: On input the secret key sk associated to tag τ and a string str, the puncturing algorithm outputs the updated secret key sk'. We also say (str, τ) has been punctured.
- $\mathsf{PuncTag}(\mathsf{sk}, \tau)$: On input the secret key sk associated to tag τ, the puncturing algorithm outputs the updated secret

[1]Recall that, in Bloom filter, max = n, where n denotes the number of elements to be added.

key sk' for the next period $\tau+1$. We also say τ has been punctured.
- $\mathsf{Sign}(\mathsf{sk}, m)$: On input the secret key and a message m, the signing algorithm outputs the tag τ and a signature σ.
- $\mathsf{Verify}(\mathsf{vk}, m, \tau, \sigma)$: On input the public key vk, a message m, a tag τ, and a signature σ, the verification algorithm outputs 1 if σ is a valid signature for m. Otherwise, it outputs 0.

Correctness of tag-based puncturable signature. Intuitively, it requires that (1) signing is always successful with the initial, non-punctured secret key, (2) signing fails if the signed message m with prefix m' and tag τ satisfies the condition that (m', τ) or τ has been punctured, and (3) otherwise, the probability that signing fails is bounded by some non-negligible function.

Definition 8 (Correctness)**.** *For any message m with prefix m', any tag τ, any $(\mathsf{sk}_{\mathsf{init}}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda)$, and any sequence of invocations of $\mathsf{sk} \leftarrow \mathsf{PuncStr}(\mathsf{sk}, .)$ and $\mathsf{sk} \leftarrow \mathsf{PuncTag}(\mathsf{sk}, .)$, we have*

- $\mathsf{Verify}(\mathsf{vk}, m, \tau_0, \mathsf{Sign}(\mathsf{sk}_{\mathsf{init}}, m)) = 1$, *where $\mathsf{sk}_{\mathsf{init}}$ is the initial, non-punctured secret key,*
- *If (m', τ) has been punctured or τ has been punctured, then* $\mathsf{Verify}(\mathsf{vk}, m, \tau, \mathsf{Sign}(\mathsf{sk}, m)) = 0$,
- *Otherwise, it holds that* $\Pr[\mathsf{Verify}(\mathsf{vk}, m, \tau, \mathsf{Sign}(\mathsf{sk}, m)) = 0] \leq \mu(\ell, k)$, *where $\mu(\cdot)$ is some (possibly non-negligible) bound.*

Security Definition. For the security definition of tag-based puncturable signature Σ, we use the following experiment to describe it. Formally, for any PPT adversary $\mathcal{A}$, we consider the experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{tps}}(1^\lambda)$ between adversary $\mathcal{A}$ and challenger $\mathcal{C}$:

1) **Setup**: The challenger computes $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Setup}(1^\lambda)$ and sends vk to adversary $\mathcal{A}$. $\mathcal{C}$ initializes three empty sets $Q_{\mathsf{sig}} = \emptyset$, $P_{\mathsf{str}} = \emptyset$ and $P_{\mathsf{tag}} = \emptyset$.
2) **Query Phase**: Proceeding adaptively, adversary A can submit the following three kinds of queries:
   - **Signature query**: On input message m from adversary $\mathcal{A}$, the challenger computes $(\tau, \sigma) \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$ and updates $Q_{\mathsf{sig}} = Q_{\mathsf{sig}} \cup \{(m, \tau)\}$. $\mathcal{C}$ sends back σ.
   - **PuncStr query**: On input string str from $\mathcal{A}$, $\mathcal{C}$ runs $\mathsf{sk} \leftarrow \mathsf{PuncStr}(\mathsf{sk}, str)$ and updates $P_{\mathsf{str}} = P_{\mathsf{str}} \cup \{(str, \tau)\}$, where sk is associated to tag τ.
   - **PuncTag query**: On input a tag τ from $\mathcal{A}$, $\mathcal{C}$ runs $\mathsf{sk} \leftarrow \mathsf{PuncTag}(\mathsf{sk}, \tau)$ and updates $P_{\mathsf{tag}} = P_{\mathsf{tag}} \cup \{\tau\}$.
3) **Challenge Phase**: $\mathcal{A}$ sends the challenge puncturing string m' and tag τ^* to $\mathcal{C}$, and $\mathcal{A}$ can still submit signature and puncture queries as described in **Query phase**.
4) **Corruption query**: If $(m', \tau^*) \in P_{\mathsf{str}}$ or $\tau^* \in P_{\mathsf{tag}}$, then $\mathcal{C}$ returns the current sk, and $\perp$ otherwise.
5) **Forgery**: $\mathcal{A}$ outputs a forgery tuple (m, τ^*, σ), where m has prefix m'.

We say that adversary $\mathcal{A}$ wins the experiment $\mathbf{Expt}_{\mathcal{A}}(1^\lambda)$ if

$$(m, \tau^*) \notin Q_{\mathsf{sig}} \land \mathsf{Verify}(\mathsf{vk}, m, \tau^*, \sigma) = 1.$$

Definition 9 (Unforgeability with adaptive puncturing)**.** *We say the tag-based puncturable signature scheme Σ is unforgeable with adaptive puncturing, if for any* PPT *adversary $\mathcal{A}$, the probability of winning experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{tps}}(1^\lambda)$ is $\mathsf{negl}(\lambda)$, where the probability is over the randomness of the challenger and adversary.*

B. Our Construction of Tag-Based Puncturable Signature

In this construction, we deploy a HIBS scheme based on an ordered binary hierarchy tree [18] of depth t and the Bloom filter to construct a tag-based puncturable signature scheme, which allows to use 2^t periods with the corresponding tag $\tau \in \{0,1\}^t$. The root node of tree has the label ε. The left child of a node under label d is labeled with $d0$ and the right child with $d1$. Two nodes at level t' of the tree are siblings if and only if the first $t'-1$ bits of their labels are equal. In a HIBS scheme, the identity space is $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_{t+1}$ where $\mathcal{D}_1 = \cdots = \mathcal{D}_t = \{0,1\}$, $\mathcal{D}_{t+1} = [\ell]$, and ℓ is the size of the Bloom filter. Each bit string $\tau \in \{0,1\}^t$ is associated to a leaf of the tree.

The basic idea of our construction is to compute and update secret keys by using the hierarchical key delegation property of HIBS. In more detail, we can derive keys for all positions of Bloom filter from a given HIBS-key sk_τ for tag τ by computing $\mathsf{sk}_{\tau|u} \leftarrow \mathsf{HIBDel}(\mathsf{sk}_\tau, u)$ for all $u \in [\ell]$. Once we have computed all Bloom filter keys $\mathsf{sk}_{\tau|u}$ for some $\tau \in \{0,1\}^t$ and puncturing operation at period τ has reached the maximum number, we can compute $\mathsf{sk}_{\tau+1}$ and $\mathsf{sk}_{\tau+1|u}$ for all $u \in [\ell]$ from sk_τ, and delete sk_τ. Specifically, the secret keys associated to all *right-hand siblings* of nodes that lie on the path from node τ to the root are computed, and then all secret keys associated to nodes that lie on the path from node τ to the root are deleted.

Let $\mathsf{BF} = (\mathsf{BF.Gen}, \mathsf{BF.Update}, \mathsf{BF.Check})$ be a Bloom filter for set $\mathbb{G}_1$ and HIBS = (HIBGen, HIBDel, HIBSign, HIBVerify) be a $(t+1)$-level hierarchical identity based signature scheme. The construction of tag-based puncturable signature scheme Σ = (Setup, Puncture, Sign, Verify) is the following:

- $\mathsf{Setup}(1^\lambda, \ell, k, t)$: This algorithm first runs $(\{H_j\}_{j\in[k]}, T) \leftarrow \mathsf{BF.Gen}(\ell, k)$ to generate a Bloom filter instance, and runs $(\mathsf{mpk}, \mathsf{sk}_\varepsilon) \leftarrow \mathsf{HIBGen}(1^\lambda)$ to generate a key pair. Then it obtains the HIBS secret key for the initialized tag $\tau = 0^t$ by recursively computing

  $$\mathsf{sk}_{0^u} \leftarrow \mathsf{HIBDel}(\mathsf{sk}_{0^{u-1}}, 0), \forall u \in [t].$$

  Next it computes the ℓ Bloom filter keys for 0^t as

  $$\mathsf{sk}_{0^t|u} \leftarrow \mathsf{HIBDel}(\mathsf{sk}_{0^t}, u), \forall u \in [\ell],$$

  and sets $\mathsf{sk}_{\mathsf{bloom}} = \{\mathsf{sk}_{0^t|u}, 0^t\}_{u\in[\ell]}$. Then it deletes the secret key sk_{0^t}, and computes

  $$\mathsf{sk}_{0^{u-1}1} \leftarrow \mathsf{HIBDel}(\mathsf{sk}_{0^{u-1}}, 1), \forall u \in [t]$$

  and sets $\mathsf{sk}_{\mathsf{update}} = \{\mathsf{sk}_{0^{u-1}1}\}_{u\in[t]}$. Finally, the algorithm returns $\mathsf{sk} = (T, \mathsf{sk}_{\mathsf{bloom}}, \mathsf{sk}_{\mathsf{update}})$ and $\mathsf{vk} = (\mathsf{mpk}, \{H_j\}_{j\in[k]})$.
- $\mathsf{PuncStr}(\mathsf{sk}, str)$: On input the secret key $\mathsf{sk} = (T, \mathsf{sk}_{\mathsf{bloom}}, \mathsf{sk}_{\mathsf{update}})$ and a string str, where $\mathsf{sk}_{\mathsf{bloom}}$ is associated to tag τ, the PuncStr algorithm computes $T' = \mathsf{BF.Update}$

$(\{H_j\}_{j\in[k]}, T, m)$. Then, for each $u \in [\ell]$ it defines

$$\mathsf{sk}'_{\tau|u} = \begin{cases} \mathsf{sk}_{\tau|u} & \text{if } T'[u] = 0, \\ \perp & \text{otherwise} \end{cases}$$

where $T'[u]$ denotes the u-th bit of T'. The algorithm sets $\mathsf{sk}'_{\text{bloom}} = (\mathsf{sk}'_{\tau|u}, \tau)_{u\in[\ell]}$, and returns $\mathsf{sk}' = (T', \mathsf{sk}'_{\text{bloom}}, \mathsf{sk}_{\text{update}})$.

- $\mathsf{PuncTag}(\mathsf{sk}, \tau)$: On input the secret key $\mathsf{sk} = (T, \mathsf{sk}_{\text{bloom}}, \mathsf{sk}_{\text{update}})$ and a tag τ, where $\mathsf{sk}_{\text{bloom}}$ is associated to tag τ, the PuncTag algorithm resets $T = 0^\ell$, then it computes $\mathsf{sk}_{\tau+1}$ from the keys contained in $\mathsf{sk}_{\text{update}}$ by the key delegation algorithm, and computes

$$\mathsf{sk}_{\tau+1|u} \leftarrow \mathsf{HIBDel}(\mathsf{sk}_{\tau+1}, u), \forall u \in [\ell].$$

  Finally, the algorithm updates $\mathsf{sk}'_{\text{update}}$ by computing the secret keys associated to all *right-hand siblings* of nodes that lie on the path from node $\tau + 1$ to the root, adding these corresponding keys to $\mathsf{sk}'_{\text{update}}$ and deleting all keys from $\mathsf{sk}'_{\text{update}}$ that lie on the path from node $\tau + 1$ to the root. The algorithm sets $\mathsf{sk}'_{\text{bloom}} = (\mathsf{sk}'_{\tau+1|u}, \tau + 1)_{u\in[\ell]}$ and returns $\mathsf{sk}' = (T', \mathsf{sk}'_{\text{bloom}}, \mathsf{sk}'_{\text{update}})$.
- $\mathsf{Sign}(\mathsf{sk}, m)$: On input secret key $\mathsf{sk} = (T, \mathsf{sk}_{\text{bloom}}, \mathsf{sk}_{\text{update}})$ where $\mathsf{sk}_{\text{bloom}} = (\mathsf{sk}_{\tau|u}, \tau)_{u\in[\ell]}$ and a message m with prefix m', the signing algorithm first checks whether BF.Check $(H, T, m') = 1$ and outputs $\perp$ in this case. Otherwise, note that BF.Check$(H, T, m') = 0$ means there exists at least one index $i_{j^*} \in \{i_1, \cdots, i_k\}$ such that $\mathsf{sk}_{\tau|i_{j^*}} \neq \perp$, where $i_j = H_j(m'), \forall j \in [k]$. Then it picks a random index i_{j^*} such that $\mathsf{sk}_{\tau|i_{j^*}} \neq \perp$, and computes $\sigma_S \leftarrow \mathsf{HIBSign}(\mathsf{sk}_{\tau|i_{j^*}}, m)$. It outputs $\sigma = \{\tau|i_{j^*}, \sigma_S\}$.
- $\mathsf{Verify}(\mathsf{vk}, m, \sigma)$: On input $\mathsf{vk} = (\mathsf{mpk}, \{H_j\}_{j\in[k]})$, a message m with prefix m', a signature $\sigma = \{\tau|i_{j^*}, \sigma_S\}$, the verification algorithm checks whether

$$i_{j^*} \in \{H_j(m') : j \in [k]\} \text{ and } \mathsf{HIBVerify}(\mathsf{mpk}, \tau|i_{j^*}, m, \sigma_S) = 1$$

  If both hold, the algorithm outputs 1, and 0 otherwise.

Lemma 2. *Our generic construction described above satisfies correctness (c.f. Definition 8).*

Proof: The correctness proof of our construction follows directly from the correctness of HIBS and the relevant properties of the Bloom filter. Therefore, the first requirement of Definition 8 follows directly from the correctness of HIBS (c.f. Definition 4), if the secret key is initial and non-punctured. Secondly, by the perfect completeness of Bloom filter, $\mathsf{BF.Check}(H, T, m') = 1$ if m' is punctured. Therefore the signing of the message m with prefix m' fails, moreover the signing for period τ will also fail if τ has been punctured before due to the unforgeable property of HIBS, thus the second requirement of Definition 8 holds. Finally, the correctness error of our construction is essentially identical to the false-positive probability of the Bloom filter and the third requirement of Definition 8 holds. ■

Theorem 2. *Assuming that an algorithm $\mathcal{A}$ wins in the experiment $\mathbf{Expt}^{\mathsf{tps}}_{\mathcal{A}}(1^\lambda)$ (c.f. Definition 9) to our construction* tps *with advantage ϵ_0, there exists an algorithm $\mathcal{B}$ wins in the experiment $\mathbf{Expt}^{\mathsf{hibs}}_{\mathcal{A}}(1^\lambda)$ (c.f. Definition 5) with advantage $\epsilon_1 \geq \epsilon_0$.*

Due to space limit, we present the security proof in Appendix B. The strategy of our proof is: suppose there exists an adversary $\mathcal{A}$ against the security of tag-based puncturable signature, we construct a simulator $\mathcal{B}$ that simulates an attack environment and uses the forgery from $\mathcal{A}$ to create a forgery for the HIBS scheme, which violates the unforgeability of HIBS as defined in Definition 5.

V. Puncturable Signature in Proof-of-Stake Blockchain

Before describing the application of puncturable signature in proof-of-stake blockchain, we recall some basic definitions [23][36] of proof-of-stake blockchain and secure properties [35][46] of blockchain. We assume that there are n stakeholders $U_1, \ldots, U_n$ and each stakeholder U_i possesses s_i stake and a public and secret key pair $(\mathsf{vk}_i, \mathsf{sk}_i)$. Without loss of generality, we assume that the public keys $\mathsf{vk}_1, \ldots, \mathsf{vk}_n$ are known by all system users. The protocol execution is divided in time units, called slots.

Definition 10 (Genesis Block)**.** *The genesis block B_0 contains the list of stakeholders identified by their public-keys, their respective stakes $(\mathsf{vk}_1, s_1), \ldots, (\mathsf{vk}_n, s_n)$ and auxiliary information ρ.*

Definition 11 (State, Block Proof, Block, Blockchain, Epoch)**.** *A state is a string $st \in \{0, 1\}^\lambda$. A block proof is a set of values B_π containing information that allows stakeholders to verify whether a block is valid. A block $B = (sl_j, st, d, B_{\pi,j}, \sigma_j)$[2] generated at a slot $sl_j \in \{sl_1, \cdots, sl_R\}$ contains the slot number sl_j, the current state $st \in \{0, 1\}^\lambda$, data $d \in \{0, 1\}^*$, a block proof $B_{\pi,j}$, and σ_j, a signature on $(sl_j, st, d, B_{\pi,j})$ computed under the signing key for sl_j of the stakeholder U_i generating the block.*

A blockchain relative to the genesis block B_0 is a sequence of blocks $B_1, \cdots, B_n$ associated with a strictly increasing sequence of slots for which the state st_i of B_i is equal to $H(B_{i-1})$, where H is a prescribed collision-resistent hash function. The length of a chain $len(\mathcal{C}) = n$ is its number of blocks. The block B_n is the head of the chain, denoted head($\mathcal{C}$). We treat the empty string ε as a legal chain and by convention set $head(\varepsilon) = \varepsilon$.

An epoch is a set of R adjacent slots $S = \{sl_1, ..., sl_R\}$, during which the stake distribution for selecting slot leaders remains unchanged.

Definition 12 (Properties of Blockchain)**.** *A blockchain protocol should satisfy the following three properties.*

- ***Common Prefix.*** *The chains $\mathcal{C}_1$ and $\mathcal{C}_2$ possessed by two honest parties at the onset of the slots $sl_1 < sl_2$ are such that $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$, where $\mathcal{C}_1^{\lceil k}$ denotes the chain obtained by removing the last k blocks from $\mathcal{C}_1$ and $\preceq$ denotes the prefix relation.*
- ***Chain Quality.*** *Consider any portion of length at least k of the chain possessed by an honest party at the onset of a round; the ratio of blocks originating from the adversary is at most $1 - \mu$. We call μ the chain quality coefficient.*

[2] Recall that the puncturable signature proposed in this paper supports puncturing at any position. However, for ease of presentation, slot number sl_j is defined as the prefix of the block, which maybe has different locations in specific PoS protocols.

- ***Chain Growth.*** *Consider the chains $\mathcal{C}_1$ and $\mathcal{C}_2$ possessed by two honest parties at the onset of two slots sl_1, sl_2 with sl_2 at least s slots ahead of sl_1. Then it holds that $len(\mathcal{C}_2) - len(\mathcal{C}_1) \geq \tau \cdot s$. We call τ the speed coefficient.*

A. Application in Ouroboros Paros Protocol

Ouroboros Praos [23], a proof-of-stake protocol, provides security against fully-adaptive corruption in the semi-synchronous setting, where the adversary can corrupt any stakeholders adaptively under the honest majority of stake assumption and an adversary-controlled message delivery delay unknown to the honest stakeholders is tolerated.

Their security analysis adopts the universal composability framework. The adversary can control transactions and blocks generated by corrupted parties by interacting with functionalities $\mathcal{F}_{\mathsf{DSIG}}$, $\mathcal{F}_{\mathsf{KES}}$ and $\mathcal{F}_{\mathsf{VRF}}$, where transactions are signed with a regular EUF-CMA secure signature scheme modelled by $\mathcal{F}_{\mathsf{DSIG}}$, blocks are signed with key-evolving signature scheme with forward security modelled by $\mathcal{F}_{\mathsf{KES}}$, and the leader selection process is executed locally using a special verifiable random function (VRF) modelled by $\mathcal{F}_{\mathsf{VRF}}$. The basic protocol for the static stake case denoted by $\mathcal{F}_{\mathsf{SPoS}}$ is constructed in the $\mathcal{F}_{\mathsf{INIT}}$-hybrid model, where the genesis stake distribution $\mathbb{S}_0$ and the nonce η used in $\mathcal{F}_{\mathsf{VRF}}$ are determined by the ideal functionality $\mathcal{F}_{\mathsf{INIT}}$. It is proved $\mathcal{F}_{\mathsf{SPoS}}$ can achieve common prefix, chain growth and chain quality by using the natural bookkeeping tool "forks" as in [36], and results remain true when $\mathcal{F}_{\mathsf{DSIG}}$, $\mathcal{F}_{\mathsf{KES}}$ and $\mathcal{F}_{\mathsf{VRF}}$ are replaced by their real-word implementations in the so-called real experiment. Finally, the protocol is extended to the dynamic case where the stake distribution changes over time. All the functionalities we mentioned above are defined in Appendix F.

The puncturable signature can resist LRSL attack due to the fact that the leader U in slot sl_j would update the secret signing key sk after the block proposed, and with the updated signing key the adversary cannot forge a signature at sl_j in the name of U and thus cannot re-write a new block at the position sl_j. Also note that as in [23], we also assume in this paper that honest stakeholders can do secure erasures, which is argued to be a reasonable assumption when capturing protocol security against adaptive adversaries [42].

We now present an ideal functionality $\mathcal{F}_{\mathsf{PS}}$ of puncturable signature scheme, and show any property of the protocol that we prove true in the hybrid experiment (including common prefix, chain quality and chain growth) will remain true in the setting $\mathcal{F}_{\mathsf{KES}}$ is replaced by $\mathcal{F}_{\mathsf{PS}}$. The revised static proof-of-stake protocol π'_{SPoS} is described in Appendix C. In addition, we show that $\mathcal{F}_{\mathsf{PS}}$ can be realized by basic puncturable signature construction in Section III.B. The case for tag-based puncturable signature scheme is similar, and the details are described in Appendix D.

In a high level, the ideal functionality $\mathcal{F}_{\mathsf{PS}}$ (as defined in Figure 2) allows an adversary that corrupts the signer to forge signatures only for messages whose part at given position (e.g., j-th part) having not been punctured. Our starting point for $\mathcal{F}_{\mathsf{PS}}$ is the basic signature functionality $\mathcal{F}_{\mathsf{SIG}}$ defined in [16] with the difference that the signing operation is packed together with a puncture operation and the signature verification operation lets the adversary set the response only for the signature of unpunctured message.

Theorem 3. *The improved proof-of-stake blockchain protocol π'_{SPoS} described in Appendix C still satisfies common prefix, chain quality and chain growth if $\mathcal{F}_{\mathsf{KES}}$ is replaced by $\mathcal{F}_{\mathsf{PS}}$.*

Due to space limit, we present the full proof in Appendix E. The strategy of our proof is: given the event of violating one of common prefix, chain quality and chain growth in an execution of π'_{SPoS} with access to $\mathcal{F}_{\mathsf{PS}}$ by adversary $\mathcal{A}$ and environment $\mathcal{Z}$, we can construct another adversary $\mathcal{A}'$ so that the corresponding event happens with the same probability in an execution of π_{SPoS} with access to $\mathcal{F}_{\mathsf{KES}}$ (c.f. Appendix F-C) by adversary $\mathcal{A}'$ and environment $\mathcal{Z}$, where π_{SPoS} is original protocol [23]. If the environment $\mathcal{Z}$ can distinguish a real execution with $\mathcal{A}$ and π'_{SPoS} (accessing $\mathcal{F}_{\mathsf{PS}}$) from an ideal execution, then $\mathcal{Z}$ can also distinguish a real execution with $\mathcal{A}'$ and π_{SPoS} (accessing $\mathcal{F}_{\mathsf{KES}}$) from an ideal execution.

Remark 3. The dynamic stake case can be extended as in [23]. Specifically, $\mathcal{F}_{\mathsf{INIT}}$ is replaced with a "resettable" variant to capture the grinding capabilities of the adversary by permitting him/her to select one from a family of r independent and uniformly random nonces, a resettable leaky beacon functionality is introduced such that provides a fresh nonce for each epoch to accommodate dynamic stake, and other sub-functionalities remain unchanged.

Realizing $\mathcal{F}_{\mathsf{PS}}$. Following the proof strategy of [16], in this section we will show how to translate a puncturable signature scheme Σ into a signature protocol π_Σ in the present setting and then prove that π_Σ can securely realize $\mathcal{F}_{\mathsf{PS}}$. Specifically, π_Σ protocol runs between a stakeholder U_S and other stakeholders $U_1, ..., U_n$ and proceeds based on a puncturable signature scheme Σ=(Setup, Puncture, Sign, Verify) as follows:

1) **Key Generation:** When U_S, running π_Σ, receives an input $(\mathsf{KeyGen}, sid, U_S)$, it verifies whether $sid = (U_S, sid')$ for some sid'. If not, it ignores the input. Otherwise, it runs $\mathsf{Setup}(1^\lambda)$, records the signing key (sid, U_S, sk) and sets $P = \emptyset$, and outputs $(\mathsf{VerificationKey}, sid, vk)$.
2) **Sign and Puncture:** When U_S receives an input (PSign, sid, U_S, $m = m'...$) for an sid which it owns the signing key (sid, U_S, sk), it checks whether $m' \in P$. If not, U_S runs $\mathsf{Sign}(sk, m)$ to obtain σ, runs $\mathsf{Puncture}(sk, m')$ to update the secret keys, sets $P = P \cup m'$ and outputs $(\mathsf{Signature}, sid, m, \sigma)$.
3) **Verify:** When a stakeholder U_i ($i \in [n]$) receives an input $(\mathsf{Verify}, sid, m, \sigma, vk')$, it outputs ($\mathsf{Verified}, sid, m$, $\mathsf{Verify}(vk', m, \sigma)$).

Theorem 4. *Let $\Sigma = (\mathsf{Setup}, \mathsf{Puncture}, \mathsf{Sign}, \mathsf{Verify})$ be a puncturable signature scheme, if Σ satisfies the unforgeability with adaptive puncturing as in Definition 7, then π_Σ securely realizes $\mathcal{F}_{\mathsf{PS}}$.*

Proof: Assume that π_Σ does not realize $\mathcal{F}_{\mathsf{PS}}$, i.e. there exists an environment $\mathcal{Z}$ that can tell whether it is interacting with a prescribed simulator $\mathcal{S}$ and $\mathcal{F}_{\mathsf{PS}}$, or with an adversary $\mathcal{A}$ and π_Σ. Then following the proof approach of [16] we can show $\mathcal{Z}$ can be used to construct a forger G that wins

Functionality $\mathcal{F}_{\mathsf{PS}}$

$\mathcal{F}_{\mathsf{PS}}$ interacts with a signer U_S and stakeholder U_i as follows:

Key Generation. Upon receiving a message $(\mathsf{KeyGen}, sid, U_S)$ from a stakeholder U_S, verify that $sid = (U_S, sid')$ for some sid'. If not, then ignore the request. Else, send $(\mathsf{KeyGen}, sid, U_S)$ to the adversary. Upon receiving $(\mathsf{PublicKey}, sid, U_S, v)$ from the adversary, send $(\mathsf{PublicKey}, sid, v)$ to U_S, record the entry (sid, U_S, v), and set $P = \emptyset$.

Sign and Puncture. Upon receiving a message $(\mathsf{PSign}, sid, U_S, m = m'...)$ from U_S, verify that (sid, U_S, v) is recorded for some sid and that $m' \notin P$. If not, then ignore the request. Else, send $(\mathsf{Sign}, sid, U_S, m)$ to the adversary. Upon receiving $(\mathsf{Signature}, sid, U_S, m, \sigma)$ from the adversary, verify that no entry $(m, \sigma, v, 0)$ is recorded. If it is, then output an error message to U_S and halt. Else, send $(\mathsf{Signature}, sid, m, \sigma)$ to U_S, record the entry $(m, \sigma, v, 1)$, and set $P = P \cup \{m'\}$.

Signature Verification. Upon receiving a message $(\mathsf{Verify}, sid, m = m'..., \sigma, v')$ from some stakeholder U_i do:

1) If $v' = v$ and the entry $(m, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition ensures completeness: If the public key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2) Else, if $v' = v$, the signer is not corrupted, and no entry $(m, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, v, 0)$. (This condition ensures unforgeability: If the public key v' is the registered one, the signer is not corrupted, and m is never by signed by the signer, then the verification fails.)
3) Else, if there is an entry (m, σ, v', f') recorded, then let $f = f'$. (This condition ensures consistency: All verification requests with identical parameters will result in the same answer.)
4) Else, if $m' \in P$, then let $f = 0$ and record the entry $(m, \sigma, v, 0)$. Otherwise, send $(\mathsf{Verify}, sid, m, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, \phi)$ from the adversary, let $f = \phi$ and record the entry (m, σ, v', ϕ). (This condition ensures that the adversary is only able to forge signatures of corrupted parties on messages with unpunctured prefix.)

Output $(\mathsf{Verified}, sid, m, f)$ to U_i.

Figure 2: Functionality $\mathcal{F}_{\mathsf{PS}}$

with non-negligible probability in the experiment $\mathbf{Expt}_G^{\mathsf{ps}}(1^\lambda)$ for the underlying puncturable signature scheme Σ as defined in Definition 7, which in turn violates the unforgeability with adaptive puncturing of Σ. Since $\mathcal{Z}$ can succeed for any simulator $\mathcal{S}$, it also succeeds for the following specific $\mathcal{S}$, where $\mathcal{S}$ runs a simulated copy of $\mathcal{A}$:

1) Any input from $\mathcal{Z}$ is forwarded to $\mathcal{A}$, and any outputs from $\mathcal{A}$ is returned to $\mathcal{Z}$.
2) Whenever $\mathcal{S}$ receives $(\mathsf{KeyGen}, sid, U_S)$ from $\mathcal{F}_{\mathsf{PS}}$, it proceeds as follows: if sid is not of the form (U_S, sid'), then $\mathcal{S}$ ignores this request. Otherwise, $\mathcal{S}$ runs $\mathsf{Setup}(1^\lambda)$, records the signing key (sid, U_S, sk), sets $P = \emptyset$, and outputs $(\mathsf{VeficationKey}, sid, vk)$ to $\mathcal{F}_{\mathsf{PS}}$.
3) Whenever $\mathcal{S}$ receives $(\mathsf{PSign}, sid, U_S, m = m'...)$ from $\mathcal{F}_{\mathsf{PS}}$, if there is a recorded signing key (sid, U_S, sk) and $m' \notin P$, $\mathcal{S}$ runs $\mathsf{Sign}(sk, m)$ to obtain σ, runs $\mathsf{Puncture}(sk, m')$ to obtain the update secret keys, sets $P = P \cup \{m'\}$ and outputs ($\mathsf{Signature}$, sid, m, σ) to $\mathcal{F}_{\mathsf{PS}}$. Otherwise, it ignores the request.
4) Whenever $\mathcal{S}$ receives $(\mathsf{Verify}, sid, m, \sigma, vk')$ from $\mathcal{F}_{\mathsf{PS}}$, it returns $(\mathsf{Verified}, sid, m, \mathsf{Verify}(vk', m, \sigma))$ to $\mathcal{F}_{\mathsf{PS}}$.
5) When $\mathcal{A}$ corrupts a party U_i, $\mathcal{S}$ corrupts U_i in the ideal world. If U_i is the signer U_S, $\mathcal{S}$ reveals the current signing keys sk and the internal state of algorithm Sign (if there exists) as the internal state of U_i.

Recall that $\mathcal{Z}$ can distinguish an ideal execution with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{PS}}$ from a real execution with $\mathcal{A}$ and π_Σ, then we would demonstrate that the underlying Σ is forgeable by constructing a forger G as follows. G runs a simulated instance of $\mathcal{Z}$, and simulates for $\mathcal{Z}$ an interaction with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{PS}}$ where G plays the role of both $\mathcal{S}$ and $\mathcal{F}_{\mathsf{PS}}$. Moreover, in the simulating process, like $\mathcal{S}$, G will also run a simulated run of $\mathcal{A}$.

When $\mathcal{Z}$ activates some party U_S with input $(\mathsf{KeyGen}, sid, U_S)$, G returns the public key vk from its experiment to $\mathcal{Z}$. When $\mathcal{Z}$ activates U_S with input $(\mathsf{Sign}, sid, U_S, m = m'...)$, G calls its signing oracle with m to obtain a signature σ, calls its puncture oracle with m' to update the secret keys, then updates the puncturing set $P = P \cup \{m'\}$ and the set of queried messages $Q_{\mathsf{sig}} = Q_{\mathsf{sig}} \cup \{m\}$. When $\mathcal{Z}$ activates an uncorrupted party with input $(\mathsf{Verify}, sid, m = m'..., \sigma, vk')$, G tests whether $m \in Q_{\mathsf{sig}}$, the signer is uncorrupted before m' is punctured, and $\mathsf{Verify}(vk', m, \sigma) = 1$. If these conditions are met, then in its experiment $\mathbf{Expt}_G^{\mathsf{ps}}(1^\lambda)$, G outputs m' as the challenge string, and makes series of queries as in Definition 7. Eventually G outputs the tuple (m, σ), succeeding in the experiment.

Denote by E the event that in a run of π_Σ with $\mathcal{Z}$ and $sid = (U_S, sid')$, the signer U_S generates a public key vk, and some party U_i is activated with a verification request $(\mathsf{Verify}, sid, m{=}m'..., \sigma, vk)$, where $\mathsf{Verify}(vk, m, \sigma) = 1$, $m \notin Q_{\mathsf{sig}}$, and U_S is not corrupted before m' is punctured. If event E does not occur, $\mathcal{Z}$ would not distinguish the between an ideal and a real executions. However, we are guaranteed that $\mathcal{Z}$ can distinguish real from ideal executions with non-negligible advantage, then event E also happens with non-negligible advantage. Note that, from the view of $\mathcal{Z}$, the interaction with G looks the same as the interaction with π_Σ, which means that whenever E happens, G outputs a successful forgery. ■

B. Applications in Other Proof-of-Stake Protocols

As we have described above, most existing proof-of-stake blockchain protocols are vulnerable to the LRSL attack, and we would show that our puncturable signature construction can also be applied in other Proof-of-stake blockchain protocols to resist LRSL attack.

In both Ouroboros [36] and Snow White [22] protocols, each block is signed by the leader using ordinary signature scheme and thus they cannot resist LRSL attack. Fortunately, their signature schemes can also be replaced by puncturable signatures directly. Specifically, in Ouroborous, the leader U_i signs the block B_i by $\sigma = \mathsf{Sign}(\mathsf{sk}_j, (st_j, d, sl_j))$ and updates the secret key of U_i by $\mathsf{Puncture}(\mathsf{sk}_j, sl_j)$, and the case in Snow White is similar with the exception that the slot parameter is replaced with the time step t. By this means, even if an adversary $\mathcal{A}$ obtains the updated secret key, he cannot sign for other block data d' at the same slot sl_i or time step t, which furthermore avoids the forks in blockchains and LRSL attack. In addition, our puncturable signature also can be applied in Ouroboros Genesis [7] protocol similar to Section V.A.

Nevertheless, the case of the application for tag-based puncturable signature in PoS blockchain protocols is somewhat subtle. As we show in Appendix D, the tag-based scheme itself is not enough for PoS blockchain to resist LRSL attacks. Specifically, the adversary with the leaked secret key in current tag τ can forge signatures on any message with the tag $\tau' > \tau$ in the same slot, and thus construct forks at the slot. We remedy this problem by binding the slot parameter with corresponding tag, in particular, let miners maintain the current tag of all parties, which is inspired by the idea of maintaining unspent transaction outputs (UTXO). For readability, we defer a detailed description to Appendix D.

C. On tolerating a non-negligible correctness error for Proof-of-Stake Blockchain

The significant efficiency improvement of our PS construction stems partially from the relaxation of tolerating a non-negligible correctness error, which, in turn, comes from the non-negligible false-positive probability of a Bloom filter. Specifically, the correctness error in our puncturable signature construction means that the signing of message m may yield $\perp$ even though the secret key has never been punctured at the prefix m' of message m. However, the correctness error can be as small as possible by adjusting the corresponding parameters in Bloom filter (see Section II.A), which implies a trade-off between non-negligible correctness error and the size of secret key.

For proof-of-stake blockchain, it is a reasonable approach to accept a small, but non-negligible correctness error, in exchange for the huge efficiency gain. In fact, existent blockchain protocols achieve security properties (i.e. common prefix, chain quality and chain growth) with high probability instead of certainty, which means a small error probability is inherent in these protocols. Moreover, the signing error would not affect the running of the blockchain system. For instance, in Ouroboros [36], the stakeholder selected as one of the leaders in current slot can still get the reward even if his signing fails. While in Ouroboros Praos [23] and Snow White [22], some slots might have multiple slot leaders or no leader (i.e., empty slot), which means the signing error for one leader would not affect the protocol running.

D. Analysis and Comparison

For proof-of-stake blockchain application, we make a comparison between our puncturable signature and two existing forward secure signature, in terms of functionality and performance. First, puncturable signature allows each leader to generate at most one block at any slot (by puncturing at sl_i, the slot number of the current block), and thus prevents attackers from compromising leaders to mount long-range attacks. Although forward secure signature can achieve the same functionality by using different secret key for signing in each period, their performance depends on the time periods, which is unsuitable for blockchain application. More specifically, in each slot of proof-of-stake blockchain, only one stakeholder is elected as a leader to propose and sign block, which means some stakeholders may only have a chance to sign block after long slots (i.e. time periods), however, the computational cost of one signature after long time periods is almost equal to that of multiple signatures for most forward secure signature schemes. On the contrary, puncturable signature can alleviate this problem because the computation is independent of time periods.

Second, keeping on signing and verifying operation as efficient as the underlying scheme is an important goal for forward-secure signatures as well as puncturable signatures. However, except for [33], almost all existing forward secure signature schemes require longer time for signing or verifying. Particularly, [39][43] requires two ordinary verification together with several hash computations, and verification time in [4][11] even grows linearly with the number of periods T. Apparently, our construction can retain the efficiency of the underlying scheme on signing and verifying, with additional k hash functions.

Third, the key update time of [33][43] depends on T or t, which may bring undesirable consumption and even become a fatal issue for some particular applications. Specifically, in the proof-of-stake blockchain, the signer may not even do any signing within one period but he has to update the signing key as long as the current period ends, which makes the update operation a vain effort. In some other applications, the signer has to update the secret key immediately after one signing operation, leading to that the number of update operations (i.e. T) within a given validity time of the public key becomes so large that the update time is unacceptable. Despite that the general construction [39] outperforms other schemes on key update, $O(T)$ non-secret certificates storage (in publicly readable tamper-proof memory) is needed, and moreover, one update includes complete key generation and verification process, which is also undesirable. The key update in our puncturable signature construction is independent on T or t, and only needs k hash computations.

Finally, in Table I we compare the performance of our construction with that of [43] and [33][3], which are most efficient in existing forward secure signature schemes. We use t_h, t'_h, t_{m1}, t_{m2}, t_{eT}, t_p, t_{mN}, t'_{mN}, t_{eN} and t_{pt} to denote the time for computing a universal hash, a hash for $H_j(j \in [k])$ in Bloom filter, a multiplication in $\mathbb{G}_1$, a multiplication in $\mathbb{G}_2$,

[3]For [43], we adopt the worst case when evaluating key update time, otherwise it depends on log$t$ after amortization with longer secret keys (additional $|H|\log t + |N|\log|H|$ bits), and moreover the total update time still increases linearly with t. For [33], we adopt the original scheme, whose key update time depends on log$T$ after optimization with longer secret key (additional $|N|(1 + \log T)$ bits) and more expensive signing and verifying operations.

TABLE I: Efficiency comparison

	Ours	[33]	[43]
Keygen time	$l \cdot t_{m1}$	$T\lambda t_{pt} + T \cdot t'_{mN} + 3t_{eN}$	λt_{eN}
Sign time	$t_{eT} + t_{m1} + k.t'_h + t_h$	$2\ t_{eN} + t_{mN} + t_h$	$2\ t_{eN} + t_{mN} + t_h$
Verify time	$k.t'_h + t_h + t_p + t_{eT} + t_{m2}$	$2\ t_{eN} + t_{mN} + t_h$	$(4\ t_{eN} + 2\ t_{mN}) + (\log\lambda + \log t)\ t_h$
Key update time	$k.t'_h$	$T \cdot t_{eN} + T\lambda \cdot t_{pt}$	$t \cdot t_{eN}$
Secret key size	$l \cdot e^{-k\|P\|/l}\|\mathbb{G}_1\|$	$3\|\mathbb{Z}_N^*\| + \lambda + 2\log T$	$\|\mathbb{Z}_N^*\| + \lambda \cdot (\log\lambda + \log t)$
Public key size	$\|\mathbb{G}_1\|$	$2\|\mathbb{Z}_N^*\| + \log T$	λ
Signature size	$\|\mathbb{Z}_p^*\| + \|\mathbb{G}_1\|$	$\|\mathbb{Z}_N^*\| + 2\lambda + \log T$	$4\|\mathbb{Z}_N^*\| + \lambda(\log\lambda + \log t)$

TABLE II: Experimental results comparison

	Ours		[33]		[43]	
	128-bit	192-bit	128-bit	192-bit	128-bit	192-bit
Keygen time (ms)	5.19×10^3	1.11×10^4	6.92×10^4	1.62×10^5	378.88	3.40×10^3
Sign time (ms)	1.17	5.61	5.92	35.41	5.92	35.41
Verify time (ms)	4.14	23.09	5.92	35.41	11.84	70.82
Key update time (ms)	10^{-5}	10^{-5}	3.65×10^5	1.93×10^6	2.96×10^5	1.77×10^6
Secret key size[1]	$1.31\times e^{-\|P\|/1.44\times10^3}M$	$1.64\times e^{-\|P\|/1.44\times10^3}M$	$1.14KB$	$2.84KB$	$761.75B$	$1.51KB$
Public key size	$95.25B$	$119.50B$	$770.08B$	$1.88KB$	$16B$	$24B$
Signature size	$129.11B$	$169.35B$	$418.08B$	$1010.08B$	$1.87KB$	$4.32KB$

[1] The secret size decreases with puncturing operation. For 128-bit and 192-bit security level, the maximum is $1.31MB$ and $1.64M$ when $|P| = 0$, and the minimum is $0.65M$ and $0.82M$ when $|P|$ reaches its maximum (i.e., 10^3), respectively.

TABLE III: Experimental cost of each unit operation (ms)

	t_{m1}	t_{m2}	t_{eT}	t_p	t_{mN}	t'_{mN}	t_{eN}	t_{pt}	t_h	t'_h
128-bit	0.36	0.97	0.81	2.36	0.0035	0.00095	2.96	0.0054	2×10^{-5}	10^{-6}
192-bit	0.77	6.93	4.84	11.32	0.011	0.0023	17.7	0.0084		

an exponentiation in $\mathbb{G}_T$, a bilinear pairing, a multiplication in $\mathbb{Z}_N^*$, a multiplication in $\varphi(N)$, an exponentiation in $\mathbb{Z}_N^*$ and one primality test for one λ-bit number, respectively. We also denote $|\mathbb{Z}_p^*|$, $|\mathbb{Z}_N^*|$ and $|\mathbb{G}_1|$ as the bit-length of an element in $\mathbb{Z}_p^*$, an element in $\mathbb{Z}_N^*$ and an element in $\mathbb{G}_1$, respectively, where p is the order of $\mathbb{G}_1$.

The implementations are written in C using version 3 of AMCL [1] and compiled using gcc 5.4.0, and the programme runs on a Lenovo ThinkCentre M8500t computer with Ubuntu 16.04.9 (64 bits) system, equipped with a 3.40 GHz Intel Core i7-4770 CPU with 8 cores and 8GB memory. Particularly, the AMCL library recommends two types of BLS curves (i.e., BLS12 and BLS24) to support bilinear pairings, and the curves have the form $y^2 = x^3 + b$ defined over a finite field $\mathbb{F}_q$, with $b = 15$ and $|q| = 383$ for BLS12, while $b = 19$ and $|q| = 479$ for BLS24, where q is a prime. According to the analysis [8], BLS12 and BLS24 curves can provide 128-bit and 192-bit security levels respectively. For the group $\mathbb{Z}_N^*$, we choose $|N| = 3072$ and $|N| = 7680$ for 128-bit and 192-bit security levels respectively. For hash function, we choose SHA-384.[4] In addition, we assume one stakeholder can be leader for 10^3 times on average and set $n = 10^3$ in Bloom filter. Without loss of generality we assume the average probability that one stakeholder is selected as the leader in one slot is 1/100 (which is large enough in practice)[5], which means there are at least 10^5 slots in blockchain and set $T = 10^5$. We also set the error probability $pr = 1/1000$ of Bloom filter, then we can compute $\ell = -\frac{n \ln pr}{(\ln 2)^2} = 1.44 \times 10^4$ and $k = \lceil \frac{l}{n} \ln 2 \rceil = 10$. Note that t in [43] denotes the time periods elapsed, also the number of signed operations so far, so we set $t = 10^5$ to evaluate the worst case.

Table II summarizes the experiment results, where the time represents the average time for 100 runs of each operation and the experimental cost of each basic operation over recommended groups at different security levels is shown in Table III. The results show that our scheme performs better on signing and verification efficiency, significantly on key update efficiency. Moreover, our scheme has the smallest signature size, which drastically reduces the communication complexity for proof-of-stake blockchain. In addition, key generation in our scheme can be further optimized by pre-computing some

[4]The hash function $H_j (j \in [k])$ in bloom filter can be simulated by two hash functions according to the analysis in [38]. In practice, the guava library [3] by Google employs Murmur3 hash [2] for Bloom filter. For simplicity, we replace Murmur3 with SHA-384 during the test, however, our scheme would perform better using the faster Murmur3.

[5]Note that here we just choose the specific parameters to carry out the efficiency comparison. For larger n and T, the efficiency of our scheme remains unchanged except that the time for key generation and secret size would increase according to Table I, and thus the advantage of our scheme over forward secure signature schemes in the aspect of sign/verify/key update time as well as signature size still holds.

exponentiations *off-line*. However, the initial secret key size in our scheme is large due to the Bloom filter. Fortunately, the secret key size shrinks with increasing amount of signing operations and can be furthermore reduced by HIBS-based optimization (i.e., clearing and reconstructing of the bloom filter frequently). In practice, the secret keys are stored locally on personal equipments, and reducing computation complexity and communication complexity may be more important with the rapid advance of storage technology.

VI. Conclusion

Although the notion of puncturable signatures has been proposed before, this is the first work that makes it efficient enough to be deployed in practice. We proposed a construction approach based on Bloom filter, whose puncturing operation only involves a small number of efficient computations (e.g. hashing), which outperforms previous schemes by orders of magnitude. In order to further improve efficiency, we also introduced a new primitive, called tag-based puncturable signature. Then we showed a generic construction based on hierarchical identity based signature scheme, and proved its security against adaptive puncturing. Our construction and security analysis are independent of any particular instantiation of building blocks. Next, we used puncturable signature to construct practical proof-of-stake blockchain protocol resilient to LRSL attacks. Our motivation stems from the observation that LRSL attack can alter transactions history and furthermore hamper the development of proof-of-stake blockchain. Our construction allows to realize practical blockchain protocol, and experiment results show that our scheme performs significantly on communication and computation efficiency.

How to design efficient puncturable signature without Bloom filter is a worthwhile direction. We believe that puncturable signature will find applications beyond proof-of-stake blockchain protocols.

References

[1] https://github.com/milagro-crypto/amcl/tree/master/version3.

[2] https://sites.google.com/site/murmurhash/.

[3] https://github.com/google/guava/blob/master/guava-tests/test/com/google/common/hash/BloomFilterTest.java.

[4] Michel Abdalla and Leonid Reyzin. A new forward-secure digital signature scheme. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 116–129. Springer, 2000.

[5] Ross Anderson. Two remarks on public-key cryptology. In *an invited lecture at the Fourth ACM Conference on Computer and Communications Security*, 1997.

[6] Michael Backes, Sebastian Meiser, and Dominique Schröder. Delegatable functional signatures. In Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang, editors, *PKC 2016, Part I*, volume 9614 of *LNCS*, pages 357–386. Springer, Heidelberg, March 2016.

[7] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 25th ACM conference on Computer and communications security*, pages 913–930. ACM, 2018.

[8] Razvan Barbulescu and Sylvain Duquesne. Updating key size estimations for pairings. *Journal of Cryptology*, pages 1–39, 2017.

[9] Paulo SLM Barreto, Benoît Libert, Noel McCullagh, and Jean-Jacques Quisquater. Efficient and provably-secure identity-based signatures and signcryption from bilinear maps. In *International conference on the theory and application of cryptology and information security*, pages 515–532. Springer, 2005.

[10] Mihir Bellare and Georg Fuchsbauer. Policy-based signatures. In Krawczyk [40], pages 520–537.

[11] Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448. Springer, 1999.

[12] Mihir Bellare, Igors Stepanovs, and Brent Waters. New negative results on differing-inputs obfuscation. In *Proceedings of EUROCRYPT 2016*, pages 792–821. Springer, 2016.

[13] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[14] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 56–73. Springer, 2004.

[15] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Krawczyk [40], pages 501–519.

[16] Ran Canetti. Universally composable signature, certification, and authentication. In *Proceedings of 17th IEEE Computer Security Foundations Workshop*, pages 219–235. IEEE, 2004.

[17] Ran Canetti. Universally composable signature, certification, and authentication. *ieee computer security foundations symposium*, 2003:219–233, 2004.

[18] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure publickey encryption scheme. In *Proceedings of EUROCRYPT 2003*, pages 255–271. Springer, 2003.

[19] Jae Choon Cha and Jung Hee Cheon. An identity-based signature from gap Diffie-Hellman groups. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 18–30. Springer, Heidelberg, January 2003.

[20] Jing Chen and Silvio Micali. Algorand. In *arXiv:1607.01341v9*, 2017.

[21] Sherman SM Chow, Lucas CK Hui, Siu Ming Yiu, and KP Chow. Secure hierarchical identity based signature and its application. In *International Conference on Information and Communications Security*, pages 480–494. Springer, 2004.

[22] CPhil Daian, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. In *Cryptology ePrint Archive, Report 2016/919*, 2016.

[23] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Proceedings of EUROCRYPT 2018*. Springer, 2018.

[24] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-rtt key exchange. In *Proceedings of EUROCRYPT 2018*. Springer, 2018.

[25] David Derler, Sebastian Ramacher, and Daniel Slamanig. Generic double-authentication preventing signatures and a post-quantum instantiation. *provable security*, 2018:258–276, 2018.

[26] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proceedings of 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013*, pages 40–49. IEEE, 2013.

[27] Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. In *Proceedings of ASIACRYPT 2002*, pages 548–566. Springer, 2002.

[28] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[29] Matthew D Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE, 2015.

[30] Felix Gunther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-rtt key exchange with full forward secrecy. In *Proceedings of EUROCRYPT 2017*, pages 519–548. Springer, 2017.

[31] Shai Halevi, Yuval Ishai, Abhishek Jain, Ilan Komargodski, Amit Sahai, and Eylon Yogev. Non-interactive multiparty computation without correlated randomness. In *Proceedings of ASIACRYPT 2017*, pages 181–211. Springer, 2017.

[32] ISO/IEC. IT security techniques – digital signatures with appendix – part 3: Discrete logarithm based mechanisms. *ISO/IEC 14883-3:2018*.

[33] Gene Itkis and Leonid Reyzin. Forward-secure signatures with optimal signing and verifying. In *Proceedings of CRYPTO 2001*, pages 332–354. Springer, 2001.

[34] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and t-pake in the password-only model. *international conference on the theory and application of cryptology and information security*, 2014:233–253, 2014.

[35] Aggelos Kiayias and Giorgos Panagiotakos. Speed-security tradeoffs in blockchain protocols. In *Cryptology ePrint Archive, Report 2015/1019*, 2015.

[36] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Proceedings of CRYPTO 2017*, pages 357–388. Springer, 2017.

[37] Eike Kiltz, Anton Mityagin, Saurabh Panjwani, and Barath Raghavan. Append-only signatures. *international colloquium on automata languages and programming*, 2005:434–445, 2005.

[38] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Structures and Algorithms*, 33(2):187–218, 2008.

[39] Hugo Krawczyk. Simple forward-secure signatures from any signature scheme. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 108–115. ACM, 2000.

[40] Hugo Krawczyk, editor. *PKC 2014*, volume 8383 of *LNCS*. Springer, Heidelberg, March 2014.

[41] Wenting Li, Sebastien Andreina, Jens-Matthias Bohli, and Ghassan Karame. Securing proof-of-stake blockchain protocols. In *Proceedings of ESORICS 2017 International Workshops, DPM2017/CBT2017*, pages 297–315. Springer, 2017.

[42] Andrew Y Lindell. Adaptively secure two-party computation with erasures. In *Cryptographers Track at the RSA Conference*, pages 117–132. Springer, 2009.

[43] Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Proceedings of EUROCRYPT 2002*, pages 400–417. Springer, 2002.

[44] Shigeo Mitsunari, Ryuichi Sakai, and Masao Kasahara. A new traitor tracing. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 85(2):481–484, 2002.

[45] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. Bitcoin and cryptocurrency technologies, 2016.

[46] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Proceedings of EUROCRYPT 2017*, pages 643–673. Springer, 2017.

[47] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

APPENDIX A
Security Proof of Theorem 1

In order to prove the security of our scheme, we consider a particular adversary $\mathcal{B}$ with fixed position against our signature scheme in a variant of the above experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{ps}}(1^\lambda)$, denoted by $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$. Specifically, in the **Setup**, the challenger returns system parameters together with a fixed position $i^* \in [l]$; the following **Query Phase** remains unchanged and the **Challenge Phase** can be omitted; in the **Corruption** query, the challenger only returns the current secret key by excluding the key at position i^*. We say $\mathcal{B}$ wins the experiment $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$, if $\mathcal{B}$ outputs some message m together with a valid signature (h, S, i^*) on m.

We sketch the proof in two steps as in [9][19]. First, we prove there exists an algorithm $\mathcal{B}$ that wins in the $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$ experiment with non-negligible advantage, if adversary $\mathcal{A}$ has non-negligible advantage against our signature scheme ps in the $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{ps}}(1^\lambda)$ experiment (c.f. Lemma 3). Then, assuming the existence of $\mathcal{B}$, we can construct an algorithm $\mathcal{C}$ that breaks the τ-SDH assumption (c.f. Lemma 4).

Lemma 3. *Assuming that an algorithm $\mathcal{A}$ wins in the $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{ps}}(1^\lambda)$ experiment (c.f. Definition 7) to our construction* ps*, with probability ϵ_0 within running time t_0, there exists an algorithm $\mathcal{B}$ that wins in the $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$ experiment as described above to* ps *which has probability $\epsilon_1 \geq \epsilon_0(1-(1-1/l)^k)/k$ within a running time $t_1 \leq t_0$.*

Proof: Suppose there exists such an adversary $\mathcal{A}$, and we construct a simulator $\mathcal{B}$ that simulates an attack environment and uses $\mathcal{A}$'s forgery to win in its own $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$ experiment. The simulator $\mathcal{B}$ can be described as follows:

- **Invocation.** $\mathcal{B}$ is invoked on a given position $i^* \in [l]$.
- **Queries.** $\mathcal{B}$ answers adaptive queries from $\mathcal{A}$ as follows:
  - $\mathcal{B}$ makes Setup query and forwards all the returned parameters to $\mathcal{A}$ for $\mathcal{A}$'s Setup query.
  - Before $\mathcal{A}$ outputs the challenge string, $\mathcal{B}$ just forwards the queries of $\mathcal{A}$, including Sign, Puncture, h_1 and h_2, to its experiment and returns the result to $\mathcal{A}$.
  - When $\mathcal{A}$ outputs the challenge string denoted by m' after the series of queries, $\mathcal{B}$ checks whether $i^* \in \{H_j(m') : j \in [k]\}$ and aborts if this does not hold. Otherwise, $\mathcal{B}$ provides the simulation for $\mathcal{A}$ as follows. For the queries h_1, h_2, Puncture and Sign, $\mathcal{B}$ just passes them to its challenger and returns the result as before. While for Corruption query, $\mathcal{B}$ firstly checks whether $m' \in P$ and returns $\emptyset$ if this does not hold. Otherwise, $\mathcal{B}$ makes Corruption query in its experiment, and returns the response sk to $\mathcal{A}$.

Eventually, $\mathcal{A}$ outputs a valid signature $(m, \sigma = (h, S, j^*))$, where m' is the prefix of m. If $j^* = i^*$, then $\mathcal{B}$ sets (m, σ) as its own output and apparently $\mathcal{B}$ also wins in its $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$ experiment.

In the simulation described above, there are two events that causes $\mathcal{B}$ to abort: (1) $i^* \notin \{H_j(m^*) : j \in [k]\}$ for the challenge string m^*; (2) $i^* \neq j^*$ for the forged signature $\sigma = (h, S, j^*)$.

Recall that the k hash functions in Bloom filter are sampled universally and independently, and thus each position in array is selected with equal probability. Besides, i^* is invisible and looks random to $\mathcal{A}$, then the selection of m' is independent of i^*. Therefore the probability that $i^* \notin \{H_j(m') : j \in [k]\}$ is $(1-1/\ell)^k$. Similarly, the second event $i^* \neq j^*$ happens with probability $1-1/k$. Combing these, with probability $\epsilon_1 \geq \epsilon_0(1-(1-1/l)^k)/k$, $\mathcal{B}$ completes the whole simulation without aborting and wins in the $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$ experiment. ∎

Lemma 4. *Assuming that an algorithm $\mathcal{B}$ wins in the $\mathbf{Expt}_{\mathcal{A}}^{\mathsf{fps}}(1^\lambda)$ experiment to our construction* ps*, with advantage $\epsilon_1 \geq 10(q_S+1)(q_S+q_{h_2})/p$ within running time t_1, there exists an algorithm $\mathcal{C}$ that breaks τ-SDH assumption for $\tau = q_{h_1}$ within running time $t_2 \leq 120686 q_{h_2} t_1/(\epsilon_1(1-\tau/p))$, where q_{h_1}, q_{h_2} and q_S are maximum query times of hash function h_1, h_2 and signing respectively.*

Proof: Suppose there exists such an adversary $\mathcal{B}$, and we construct a simulator $\mathcal{C}$ that simulates an attack environment and uses $\mathcal{B}$'s forgery to break the τ-SDH assumption. The simulator $\mathcal{C}$ can be described as follows:

- **Invocation.** $\mathcal{C}$ takes as input a random instance (P_1, P_2, αP_2, $\alpha^2 P_2$,...,$\alpha^\tau P_2$) and aims to find a pair (h, $\frac{1}{h+\alpha}P_1$) for some $h \in Z_p^*$.
- **Setup.** $\mathcal{C}$ samples $\tau - 1$ elements w_1,w_2,...,$w_{\tau-1} \xleftarrow{R} Z_p^*$, where the w_i ($i \in [\tau-1]$) will be used as the response to $\mathcal{B}$'s h_1 queries. $\mathcal{C}$ expands the polynomial $f(y) = \prod_{i=1}^{\tau-1}(y+w_i)$ to obtain $c_0, c_1, ..., c_{\tau-1} \in Z_p^*$ so that $f(y) = \prod_{i=0}^{\tau-1} c_i y^i$. Then $\mathcal{C}$ sets $\hat{P}_2 = \sum_{i=0}^{\tau-1} c_i(\alpha^i)P_2 = f(\alpha)P_2 \in \mathbb{G}_2$ and $\hat{P}_1 = \psi(\hat{P}_2) \in \mathbb{G}_1$ to be new generators of $\mathbb{G}_2$ and $\mathbb{G}_1$. Then the master public key is set to $\hat{P}_{pub} = \sum_{i=1}^{\tau} c_{i-1}(\alpha^i P_2)$ such that $\hat{P}_{pub} = \alpha\hat{P}_2$, thus the master secret key is the unknown α.
  To provide the secret keys corresponding to the positions having been queried to h_1, $\mathcal{C}$ expands $f_i(y) = f(y)/(y + w_i) = \sum_{i=0}^{\tau-2} d_i y^i$ and
  $$\sum_{i=0}^{\tau-2} d_i\psi(\alpha^{i+1}P_2) = \alpha f_i(\alpha)P_1 = \frac{\alpha f(\alpha)P_1}{\alpha + w_i} = \frac{\alpha}{\alpha + w_i}\hat{P}_1 \tag{1}$$
  Thus, the $\tau - 1$ pairs (w_i, $\frac{\alpha}{\alpha+w_i}\hat{P}_1$) can be computed using the left member of equation (1).
  Then $\mathcal{C}$ provides the parameters (p, e, ψ, $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, $\hat{P}_1$, $\hat{P}_2$, h_1, h_2) to $\mathcal{B}$. In addition, $\mathcal{C}$ also generates a Bloom filter as $(\{H_j\}_{j\in[k]}, T) \leftarrow \mathsf{BF.Gen}(\ell, k)$, then outputs the verification key $\mathsf{vk} = (\hat{P}_{pub}, g, \{H_j\}_{j\in[k]})$ (now $g = e(\hat{P}_1, \hat{P}_{pub})$) and challenge position i^* to $\mathcal{B}$. Then $\mathcal{C}$ is ready to answer $\mathcal{B}$'s queries during the simulation. For simplicity, we may assume as in [19] that for any $i \in [l]$, $\mathcal{B}$ queries $h_1(i)$ at most once and any query involving i is proceeded by the RO query $h_1(i)$, which means that $\mathcal{B}$ has to query $h_1(i)$ before he can obtain a signature (h, S, i) from **Signature** query and obtain the secret key sk_i from **Corruption** query by using a simple wrapper of $\mathcal{B}$.
- **Hash function queries.** $\mathcal{C}$ initializes a counter t to 1.
  - $h_1 : [l] \rightarrow \mathbb{Z}_p^*$: On input $i \in [l]$, $\mathcal{C}$ returns a random $w = w^* \xleftarrow{R} \mathbb{Z}_p^*$ if $i = i^*$. Otherwise, $\mathcal{C}$ returns $w = w_t$ and increments t. In both cases, $\mathcal{C}$ stores (i, w) in a list L_1.
  - $h_2 : \{0,1\}^* \times \mathbb{G}_T \rightarrow \mathbb{Z}_p^*$: On input (m, r), $\mathcal{C}$ outputs h if (m, r, h) is in the list L_2 (initialized to be empty). Otherwise, it outputs a random element h and stores (m, r, h) in L_2.

  Note that, according to the query of h_1 and the computation in the setup phase, $\mathcal{C}$ knows the secret key $sk_i = \frac{\alpha}{\alpha+w_i}\hat{P}_1$ for $i \neq i^*$.
- **Query Phase.** $\mathcal{C}$ answers adaptive signing and puncturing queries from $\mathcal{B}$ as follows:
  - **Signature query**: On input a message m with prefix m', $\mathcal{C}$ first checks $\mathsf{BF.Check}(\{H_j\}_{j\in[k]}, T, m') = 1$ and outputs $\perp$ in this case. Otherwise, there exists at least one index $i_j \in \{i_1, \ldots, i_k\}$ such that $\mathsf{sk}_{i_j} \neq \perp$, where $i_j = H_j(m')$, for $j \in [k]$. $\mathcal{C}$ picks a random $j \in \{i_1, \ldots, i_k\}$, $S \xleftarrow{R} G$ and $h \xleftarrow{R} \mathbb{Z}_p^*$, computes and sets $r = e(S, h_1(j)\hat{P}_2 + \hat{P}_{pub}).e(\hat{P}_1, \hat{P}_{pub})^h$, and backpatches to define $h_2(m, r) = h$. Finally, $\mathcal{C}$ stores (m, r, h) in L_2, and returns the signature $\sigma = (h, S, j)$ ($\mathcal{C}$ aborts in the unlikely collision event that $h_2(m, r)$ is already defined by other query results of Sign or h_2, the probability of which is negligible since r is random [47]).
  - **Puncture query**: On input a string str, $\mathcal{C}$ first updates $T' = \mathsf{BF.Update}(\{H_j\}_{j\in[k]}, T, str)$. Then for each $i \in [\ell]$, update
  $$\mathsf{sk}'_i = \begin{cases} \mathsf{sk}_i, & \text{if } T'[i] = 0 \\ \perp, & \text{otherwise} \end{cases}$$
  The updated signing key is $\mathsf{sk}' = (T', \{\mathsf{sk}'_i\}_{i\in[\ell]})$.
- **Corruption query**: For Corruption query, $\mathcal{C}$ recovers the matching pair (i, w) from L_1 and returns the previously computed $\frac{\alpha}{\alpha+w}\hat{P}_1$.
- **Forgery.** If adversary $\mathcal{B}$ forges a valid tuple (m, r, h, S, i^*) in a time t_1 with probability $\epsilon_1 \geq 10(q_S + 1)(q_S + q_{h_2})/p$, where the message m has prefix m', according to the forking lemma [47], $\mathcal{C}$ can replay adversary $\mathcal{B}$ with different choices of random elements for hash function h_2 and obtain two valid tuples (m, r, h', S_1, i^*) and (m, r, h'', S_2, i^*), with $h' \neq h''$ in expected time $t_2 \leq 120686 q_{h_2} t_1/\epsilon_1$.

Now a standard argument for outputs of the forking lemma can be applied as follows: $\mathcal{C}$ recovers the pair (i^*, w^*) from L_1, and note that $w^* \neq w_1, ..., w_{\tau-1}$ with probability at least $1-\tau/p$. Since both forgeries satisfies the verification equation, we can obtain the following relations:

$$e(S_1, Q_{i^*}).e(\hat{P}_1, \hat{P}_{pub})^{h'} = e(S_2, Q_{i^*}).e(\hat{P}_1, \hat{P}_{pub})^{h''},$$

where $Q_{i^*} = h_1(i^*)\hat{P}_2 + \hat{P}_{pub} = (w^* + \alpha)\hat{P}_2$. Then we have

$$e((h'' - h')^{-1} \cdot (S_1 - S_2), Q_{i^*}) = e(\hat{P}_1, \hat{P}_{pub}),$$

and hence $T^* = (\hat{P}_1 - (h'' - h')^{-1}.(S_1 - S_2))/w^* = (\hat{P}_1 - \frac{\alpha}{w^*+\alpha}\hat{P}_1)/w^* = \frac{1}{w^*+\alpha}\hat{P}_1 = \frac{f(\alpha)}{w^*+\alpha}P_1$. From T^*, $\mathcal{C}$ can proceed as in [14] to extract $\sigma^* = \frac{1}{w^*+\alpha}P_1$: $\mathcal{C}$ first writes the polynomial f as $f(y) = \gamma(y)(y + w^*) + \gamma_{-1}$ for some polynomial $\gamma(y) = \sum_{i=0}^{\tau-2}\gamma_i y^i$ and some $\gamma_{-1} \in \mathbb{Z}_p^*$ by using long division method, and eventually computes

$$\sigma^* = \frac{1}{\gamma_{-1}}\big[T^* - \sum_{i=0}^{\tau-2}\gamma_i\psi(\alpha^i P_2)\big] = \frac{1}{w^* + \alpha}P_1$$

and returns (w^*, σ^*) as the solution to the τ-SDH instance. ■

The combination of the above lemmas yields Theorem 1.

Appendix B
Security Proof of Theorem 2

Proof: The proof we show below is a straightforward reduction from the security of tag-based PS to the counterpart of HIBS. Suppose there exists an adversary $\mathcal{A}$ against the security of tag-based PS, we construct a simulator $\mathcal{B}$ that simulates an attack environment and uses the forgery from $\mathcal{A}$ to create a forgery for the HIBS scheme. The simulator $\mathcal{B}$ can be described as follows:

- **Invocation.** $\mathcal{B}$ obtains mpk of the HIBS scheme, and is required to return a forgery satisfying the conditions in Definition 5.

- **Setup.** $\mathcal{B}$ sets up a Bloom filter $(\{H_j\}_{j\in[k]}, T) \leftarrow \mathsf{BF.Gen}(\ell, k)$, and initializes the tag $\tau = 0^t$ and $Q_{\mathsf{sign}} = \emptyset$. $\mathcal{B}$ sends $\mathsf{vk} = (\mathsf{mpk}, \{H_j\}_{j\in[k]})$ to adversary $\mathcal{A}$.
- **Query Phase.** $\mathcal{B}$ answers adaptive signing and puncturing queries from adversary $\mathcal{A}$ as
  - **Signature query**: On input message m with prefix m' from adversary $\mathcal{A}$, $\mathcal{B}$ first checks whether $\mathsf{BF.Check}(H, T, m') = 1$ and outputs $\perp$ in this case. Otherwise, note that $\mathsf{BF.Check}(H, T, m') = 0$ means there exists at least one index $i_{j^*} \in \{i_1, ..., i_k\}$ such that $T[i_{j^*}] = 0$, where $i_j = H_j(m')(j \in [k])$. Then it picks a random index i_{j^*} such that $T[i_{j^*}] = 0$, and obtains signature σ by querying $(\tau|i_{j^*}, m)$ the signing oracle of HIBS. $\mathcal{B}$ sends $(\tau|i_{j^*}, \sigma)$ to adversary $\mathcal{A}$ and adds (m, τ) to set Q_{sig}.
  - **PuncStr query**: On input a string str, $\mathcal{B}$ updates T=$\mathsf{BF.Update}$ $(\{H_j\}_{j\in[k]}, T, str)$ and $P_{\mathsf{str}} = P_{\mathsf{str}} \cup \{(str, \tau)\}$.
  - **PuncTag query**: On input tag τ, $\mathcal{B}$ resets $T = 0^l$, updates $P_{\mathsf{tag}} = P_{\mathsf{tag}} \cup \{\tau\}$ and $\tau = \tau + 1$.
- **Challenge Phase**: $\mathcal{A}$ sends the challenge puncturing (m', τ^*) to $\mathcal{B}$, and can still submit the above queries to $\mathcal{B}$.
- **Corruption query**: If $(m', \tau^*) \in P_{\mathsf{str}}$ or $\tau^* \in P_{\mathsf{tag}}$, then $\mathcal{B}$ sends the current sk to adversary $\mathcal{A}$ using the $\mathsf{Delegate}$ oracle in HIBS security experiment.
- **Forgery**: Adversary $\mathcal{A}$ outputs forgery tuple $(m = m'..., \tau^*|i^*, \sigma_S)$, such that $(m, \tau^*) \notin Q_{\mathsf{sig}}$ and $\mathsf{Verify}(\mathsf{mpk}, m, \tau^*, \sigma) = 1$.

According to the corruption rules, adversary $\mathcal{A}$ does not obtain secret keys corresponding to identity $\tau^*|i^*$, so the tuple $(m, \tau^*|i^*, \sigma_S)$ is a valid forgery corresponding to the identity $\tau^*|i^*$ in HIBS. This concludes the proof. ■

Appendix C
Revised Ouroboros Praos Protocol π'_{SPoS}

In Figure 3, we describe the revised protocol π'_{SPoS}, where we replace the functionality $\mathcal{F}_{\mathsf{KES}}$ in the original π_{SPoS} with the $\mathcal{F}_{\mathsf{PS}}$ proposed in this paper to resist LRSL attacks, and others remain unchanged.

Appendix D
Tag-based Puncturable Signature in Proof-of-Stake Blockchain

In this part, we show tag-based puncturable signature can be deployed in proof-of-stake blockchain under some reasonable assumption. In general, if the slot sl_j contained in signed message m is bound to one specific tag and moreover this binding relation is publicly checkable, tag-based puncturable signature can guarantee the same security as the original puncturable signature.

Note that, the tag-based PS scheme itself is not enough for PoS blockchain to resist LRSL attacks. In more detail, as described in Section IV.A, the adversary with the leaked secret key in current tag τ can forge signatures on any messages in any future tag $\tau' > \tau$. When applied to PoS blockchain protocols, assuming the leader U_i issue a new block B_i by $\sigma = \mathsf{Sign}(\mathsf{sk}_i, (sl_i, st_i, d, B_\pi))$ with the current tag τ encoded in σ, then with sk_i in tag τ the adversary can forge a valid signatures $\sigma' = \mathsf{Sign}(\mathsf{sk}'_i, (sl_i, st_i, d', B'_\pi))$ with the tag $\tau' > \tau$ in the same slot sl_i and construt a fork at sl_i, even though (sl_i, τ) has been punctured. To remedy this problem, we let all miners maintain the current tag of U_i, so that they can reject σ' if the embedded tag is not the correct τ. Next we show how to achieve this additional check in implementation.

In proof-of-stake blockchain application, if stakeholder U_i is chosen as leader in slot sl_i, he signs the block B_i by $\sigma = \mathsf{Sign}(\mathsf{sk}_i, (sl_i, st_i, d, B_\pi))$, and the current tag τ is encoded in the σ itself. The tag τ will be updated to $\tau + 1$ if and only if the signing times of U_i denoted by N_{U_i} reaches max which denotes the maximum number of puncturing times, in other words, $\tau = \lfloor N_{U_i}/\mathsf{max} \rfloor$. Then we let each user (specifically, the miner) maintains one list $\mathbb{L}$ consisting of entries (PK_{U_i}, N_{U_i}) of all users, where PK_{U_i} denotes the public key of U_i, and N_{U_i} is initialized to be 0 and updated by $N_{U_i} = N_{U_i} + 1$ once one signature on a new block issued by the leader U_i is generated. Then the signature on message m with tag τ and public key PK_{U_i} would be accepted, if and only if the Verify algorithm in TPS returns 1 and moreover $\tau = \lfloor N_{U_i}/\mathsf{max} \rfloor$ for $(PK_{U_i}, N_{U_i}) \in \mathbb{L}$. In fact, our solution is inspired by the idea of maintaining UTXO in blockchain, where fully validating nodes must maintain the entire set of UTXO [45] and each entry in UTXO has similar form indicating the available coins for one address.

By this binding, tag-based puncturable signature can perfectly achieve the property puncturable signature provides for proof-of-stake blockchain. Specifically, for the additional check, the construction of tag-based puncturable signature in Section IV.B is extended as follows:

1) As an initialization, we set the maximum of puncturing max according to the desirable error probability (i.e., $\mathsf{max} = n$, where n denotes the number of elements to be added in Bloom filter), and set $\mathbb{L} = \emptyset$.
2) The entry (PK_{U_i}, N_{U_i}) for the leader U_i in the public $\mathbb{L}$ is updated by the miners after U_i generates one block by $\mathsf{Sign}()$ algorithm.
3) The verification algorithm is renewed as follows. On input $\mathsf{vk} = (\mathsf{mpk}, \{H_j\}_{j\in[k]})$, a message m with prefix m', a signature $\sigma = \{\tau|i_{j^*}, \sigma_S\}$, it outputs 1 if the following conditions hold: (1) $\tau = \lfloor N_{U_i}/\mathsf{max} \rfloor$, (2) $i_{j^*} \in \{H_j(m') : j \in [k]\}$, and (3) $\mathsf{HIBVerify}(\mathsf{mpk}, \tau|i_{j^*}, m, \sigma_S) = 1$. Otherwise, it outputs 0.

Since only a publicly verifiable check is added, the security property of tag-based PS schemes in Section IV.B still holds and can guarantee that the adversary cannot forge signatures at the punctured slot. Then we present an ideal functionality $\mathcal{F}_{\mathsf{TPS}}$ of tag-based puncturable signature scheme in Figure 4 and show any property of the protocol that we prove true in the hybrid experiment (including common prefix, chain quality and chain growth) will remain true in the setting $\mathcal{F}_{\mathsf{KES}}$ is replaced by $\mathcal{F}_{\mathsf{TPS}}$. In addition, we show that $\mathcal{F}_{\mathsf{TPS}}$ can be realized securely by the tag based puncturable signature. The details are similar to those in Section V.A, and we omit them here.

Appendix E
Security Proof of Theorem 3

Proof: Given the event of violating one of common prefix, chain quality and chain growth in an execution of π'_{SPoS} with

Protocol π'_{SPoS}

The protocol π'_{SPoS} is run by stakeholders, initially equal to $U_1, \cdots, U_n$ interacting among themselves and with ideal functionalities $\mathcal{F}_{\text{INIT}}$, $\mathcal{F}_{\text{VRF}}$, $\mathcal{F}_{\text{PS}}$, $\mathcal{F}_{\text{DSIG}}$, H over a sequence of slots $S = \{sl_1, \cdots, sl_r\}$. Define $T_i \triangleq 2^{\ell_{\text{VRF}}}\phi_f(\alpha_i)$ as the threshold for a stakeholder U_i, where α_i is the relative stake of stakeholder U_i, ℓ_{VRF} denotes the output length of $\mathcal{F}_{\text{VRF}}$, f is the active slots coefficient and $\phi_f(\alpha_i) = 1 - (1 - f)^{\alpha_i}$.

Then π'_{DPoS} proceeds as follows:

Initialization. The stakeholder U_i sends $(\mathsf{KeyGen}, sid, U_i)$ to $\mathcal{F}_{\text{VRF}}$, $\mathcal{F}_{\text{PS}}$ and $\mathcal{F}_{\text{DSIG}}$; receiving (PublicKey, sid, v_i^{vrf}), (PublicKey, sid, v_i^{ps}) and (PublicKey, sid, v_i^{dsig}), respectively. Then, in case it is the first round, it sends (ver_keys, $sid, U_i, v_i^{\text{vrf}}, v_i^{\text{ps}}, v_i^{\text{dsig}}$) to $\mathcal{F}_{\text{INIT}}$ (to claim stake from the genesis block). In any case, it terminates the round by returning $(U_i, v_i^{\text{vrf}}, v_i^{\text{ps}}, v_i^{\text{dsig}})$ to $\mathcal{Z}$. In the next round, it sends (genblock_req, sid, U_i) to $\mathcal{F}_{\text{INIT}}$, receiving (genblock, $sid, \mathbb{S}_0, \eta$) as the answer. If U_i is initialized in the first round, it sets the local blockchain $\mathcal{C} = B_0 = (\mathbb{S}_0, \eta)$ and its initial internal state $st = H(B_0)$. In case U_i is initialized after the first round, it sets its initial state to $st = H(\text{head}(\mathcal{C}))$ where $\mathcal{C}$ is the initial local chain provided by the environment.

Chain Extension. After initialization, for every slot $sl_j \in S$, every online stakeholder U_i performs the following steps:

1) U_i receives from the environment the transaction data $d \in \{0,1\}^*$ to be inserted into the blockchain.
2) U_i collects all valid chains received via diffusion into a set $\mathbb{C}$, pruning blocks belonging to future slots and verifying that for every chain $\mathcal{C}' \in \mathbb{C}$ and every block $B' = (st', d', sl', B'_\pi, \sigma_{j'}) \in \mathcal{C}'$ it holds that the stakeholder who created it is in the slot leader set of slot sl' (by parsing B'_π as (U_s, y', π') for some s, verifying that $\mathcal{F}_{\text{VRF}}$ responds to (Verify, $sid, \eta \| sl', y', \pi', v_s^{\text{vrf}}$) by (Verified, $sid, \eta \| sl', y', \pi', 1$), and that $y' < T_s$), and that $\mathcal{F}_{\text{PS}}$ responds to (Verify, $sid, (sl', st', d', B'_\pi), \sigma_{j'}, v_s^{\text{ps}}$) by (Verified, $sid, (sl', st', d', B'_\pi), 1$). U_i computes $\mathcal{C}' = \mathsf{maxvalid}(\mathcal{C}, \mathbb{C})$, sets $\mathcal{C}'$ as the new local chain and sets state $st = H(\text{head}(\mathcal{C}'))$.
3) U_i sends (EvalProve, $sid, \eta \| sl_j$) to $\mathcal{F}_{\text{VRF}}$, receiving (Evaluated, sid, y, π). U_i checks whether it is in the slot leader set of slot sl_j by checking that $y < T_i$. If yes, it generates a new block $B = (sl_j, st, d, B_\pi, \sigma)$ where st is its current state, $d \in \{0,1\}^*$ is the transaction data, $B_\pi = (U_i, y, \pi)$ and σ is a signature obtained by sending (PSign, $sid, U_i, (sl_j, st, d, B_\pi)$) to $\mathcal{F}_{\text{PS}}$ and receiving (Signature, $sid, (sl_j, st, d, B_\pi), \sigma$). U_i computes $\mathcal{C}' = \mathcal{C}|B$, sets $\mathcal{C}'$ as the new local chain and sets state $st = H(\text{head}(\mathcal{C}'))$. Finally, if U_i has generated a block in this step, it diffuses $\mathcal{C}'$.

Signing Transactions. Upon receiving (sign_tx, sid', tx) from the environment, U_i sends (Sign, sid, U_i, tx) to $\mathcal{F}_{\text{DSIG}}$, receiving (Signature, sid, tx, σ). Then, U_i sends (signed_tx, sid', tx, σ) back to the environment.

Figure 3: Protocol π'_{SPoS}

access to $\mathcal{F}_{\text{PS}}$ by adversary $\mathcal{A}$ and environment $\mathcal{Z}$, we can construct an adversary $\mathcal{A}'$ so that the corresponding event happens with the same probability in an execution of π_{SPoS} with access to $\mathcal{F}_{\text{KES}}$ (c.f. Appendix F-C) by adversary $\mathcal{A}'$ and environment $\mathcal{Z}$, where π_{SPoS} is original protocol [23]. Specifically, the adversary $\mathcal{A}'$ simulates $\mathcal{A}$ as follows:

- Upon receiving $(\mathsf{KeyGen}, sid, U_S)$ from $\mathcal{F}_{\text{PS}}$, $\mathcal{A}'$ runs as in the case of $\mathcal{F}_{\text{KES}}$ for key generation, sets counter $\mathsf{k}_{\text{ctr}} = 1$ and $P = \emptyset$, and sends $(\mathsf{PublicKey}, sid, U_S, v)$ to $\mathcal{F}_{\text{PS}}$.
- Upon receiving $(\mathsf{Sign}, sid, U_S, m = m' \cdots)$ from $\mathcal{F}_{\text{PS}}$, $\mathcal{A}'$ ignores the request if $m' \in P$. Otherwise, it sets $j = \mathsf{k}_{\text{ctr}}$ and computes the signature σ as in the case of $\mathcal{F}_{\text{KES}}$. Then $\mathcal{A}'$ updates the corresponding secret key, sets counter $\mathsf{k}_{\text{ctr}} = j+1$ and $P = P \cup m'$, and sends $(\mathsf{Signature}, sid, U_S, m, \sigma)$ to $\mathcal{F}_{\text{PS}}$.
- Upon receiving $(\mathsf{Verify}, sid, m, \sigma, v')$ from $\mathcal{F}_{\text{PS}}$, $\mathcal{A}'$ verifies the signature as in the case of $\mathcal{F}_{\text{KES}}$, and sends $(\mathsf{Verified}, sid, m, \phi)$ to $\mathcal{F}_{\text{PS}}$.

Note that in an execution of π'_{SPoS} with access to $\mathcal{F}_{\text{PS}}$, m' in $\mathcal{F}_{\text{PS}}$ equals sl (i.e, the slot parameter of the last block) (c.f. Definition 11), while in the execution of π_{SPoS} with access to $\mathcal{F}_{\text{KES}}$, the input to signature algorithm is $(\mathsf{Usign}, sid, m = sl||..., sl)$, which means that the update of punctured set P is consistent with that of counter k_{ctr}. In other words, when one signing happens on m containing some prefix sl, P adds sl in $\mathcal{F}_{\text{PS}}$ while k_{ctr} increases by 1 in $\mathcal{F}_{\text{KES}}$.

Therefore, $\mathcal{A}'$ can simulate the execution for $\mathcal{A}$. If the environment $\mathcal{Z}$ can distinguish a real execution with $\mathcal{A}$ and π'_{SPoS} (accessing $\mathcal{F}_{\text{PS}}$) from an ideal execution that provides the properties of common prefix, chain quality and chain growth, then $\mathcal{Z}$ can also distinguish a real execution with $\mathcal{A}'$ and π_{SPoS} (accessing $\mathcal{F}_{\text{KES}}$) from an ideal execution, which means that any winning advantage of the adversary against common prefix, chain quality and chain growth in π'_{SPoS} with access to $\mathcal{F}_{\text{PS}}$ immediately implies at least the same advantage in π_{SPoS} with access to $\mathcal{F}_{\text{KES}}$. ■

APPENDIX F
Ideal Functionality

A. Ideal Functionality $\mathcal{F}_{\text{INIT}}$

In [23], the genesis stake distribution $\mathbb{S}_0$ and the nonce η (to be written in the genesis block B_0) are determined by the ideal functionality $\mathcal{F}_{\text{INIT}}$ which we describe in Figure 5. In addition, $\mathcal{F}_{\text{INIT}}$ also incorporates the diffuse functionality which allows for adversarially-controlled delayed delivery of messages diffused among stakeholders and would be implicitly used by all parties to send messages and keep synchronized with a global clock.

B. Ideal Functionality $\mathcal{F}_{\text{DSIG}}$

In Figure 6, we describe the ideal functionality $\mathcal{F}_{\text{DSIG}}$ as presented in [23], and it is shown that EUF-CMA secure

Functionality $\mathcal{F}_{\mathsf{TPS}}$

$\mathcal{F}_{\mathsf{PS}}$ interacts with a signer U_S and stakeholder U_i as follows:

Key Generation. Upon receiving a message $(\mathsf{KeyGen}, sid, U_S)$ from a stakeholder U_S, verify that $sid = (U_S, sid')$ for some sid'. If not, then ignore the request. Else, send $(\mathsf{KeyGen}, sid, U_S)$ to the adversary. Upon receiving $(\mathsf{PublicKey}, sid, U_S, v)$ from the adversary, send $(\mathsf{PublicKey}, sid, v)$ to U_S, record the entry (sid, U_S, v), and set $P_{\mathsf{str}} = P_{\mathsf{tag}} = \emptyset$ and $N_{U_S} = 0$.

Sign and Puncture. Denote by τ_{cur} the current tag. Upon receiving a message $(\mathsf{PSign}, sid, U_S, m = m'...)$ from U_S, verify that (sid, U_S, v) is recorded for some sid and $(m', \tau_{cur}) \notin P_{\mathsf{str}}$. If not, then ignore the request. Else, send $(\mathsf{Sign}, sid, U_S, m, \tau_{cur})$ to the adversary.

Upon receiving $(\mathsf{Signature}, sid, U_S, m, (\tau_{cur}, \sigma_S))$ from the adversary, verify that no entry $(m, (\tau_{cur}, \sigma_S), v, 0)$ is recorded. If it is, then output an error message to U_S and halt. Else, send $(\mathsf{Signature}, sid, m, (\tau_{cur}, \sigma_S))$ to U_S, record the entry $(m, (\tau_{cur}, \sigma_S), v, 1)$, and set $P_{\mathsf{str}} = P_{\mathsf{str}} \cup (m', \tau_{cur})$ and $N_{U_S} = N_{U_S} + 1$. If $N_{U_S} \% \mathsf{max} = 0$, then set $P_{\mathsf{tag}} = P_{\mathsf{tag}} \cup \{\tau_{cur}\}$ and $\tau_{cur} = \tau_{cur} + 1$, where max denotes the maximum number of puncturing times as mentioned above.

Signature Verification. Upon receiving a message $(\mathsf{Verify}, sid, m = m'..., \sigma = \{\tau, \sigma_S\}, v')$ from some stakeholder U_i do:

1) If $v' = v$ and the entry $(m, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition ensures completeness: If the public key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2) Else, if $v' = v$, the signer is not corrupted, and no entry $(m, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, v, 0)$. (This condition ensures unforgeability: If the public key v' is the registered one, the signer is not corrupted, and m is never by signed by the signer, then the verification fails.)
3) Else, if there is an entry (m, σ, v', f') recorded, then let $f = f'$. (This condition ensures consistency: All verification requests with identical parameters will result in the same answer.)
4) Else, if $\tau < \tau_{cur}$, or $\tau > \tau_{cur}$, or $\{\tau = \tau_{cur}\} \wedge \{(m', \tau_{cur}) \in P_{\mathsf{str}})\}$, then let $f = 0$ and record the entry $(m, \sigma, v', 0)$. Otherwise, send $(\mathsf{Verify}, sid, m, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, \phi)$ from the adversary, let $f = \phi$ and record the entry (m, σ, v', ϕ). (This condition ensures that the adversary is only able to forge signatures of corrupted parties on messages with unpunctured prefix in period with correct tag.)

Output $(\mathsf{Verified}, sid, m, f)$ to U_i.

Figure 4: Functionality $\mathcal{F}_{\mathsf{TPS}}$

Functionality $\mathcal{F}_{\mathsf{INIT}}$

$\mathcal{F}_{\mathsf{INIT}}$ incorporates the delayed diffuse functionality and is parameterized by the number of initial stakeholders n and their respective stakes $s_1, ..., s_n$. $\mathcal{F}_{\mathsf{INIT}}$ interacts with the stakeholders $U_1, ..., U_n$ as follows:

– In the first round, upon a request from some stakeholder U_i of the form $(\mathsf{ver\_keys}, sid, U_i, v_i^{\mathsf{vrf}}, v_i^{\mathsf{ps}}, v_i^{\mathsf{dsig}})$, it stores the public keys tuple $(U_i, v_i^{\mathsf{vrf}}, v_i^{\mathsf{ps}}, v_i^{\mathsf{dsig}})$ and acknowledges its receipt. If any of the n stakeholders does not send a request of this form to $\mathcal{F}_{\mathsf{INIT}}$, it halts. Otherwise, it samples and stores a random value $\eta \xleftarrow{\$} \{0,1\}^\lambda$ and constructs a genesis block $(\mathbb{S}_0, \eta)$, where $\mathbb{S}_0 = \big((U_1, v_1^{\mathsf{vrf}}, v_1^{\mathsf{ps}}, v_1^{\mathsf{dsig}}, s_1), ..., (U_n, v_n^{\mathsf{vrf}}, v_n^{\mathsf{ps}}, v_n^{\mathsf{dsig}}, s_n)\big)$.

– In later rounds, upon a request of the form $(\mathsf{genblock\_req}, sid, U_i)$ from some stakeholder U_i from some stakeholder U_i, $\mathcal{F}_{\mathsf{INIT}}$ sends $(\mathsf{genblock}, sid, \mathbb{S}_0, \eta)$ to U_i.

Figure 5: Functionality $\mathcal{F}_{\mathsf{INIT}}$

signature schemes realize $\mathcal{F}_{\mathsf{DSIG}}$ in [17]. This functionality is used to model signatures on transactions in this paper.

C. Ideal Functionality $\mathcal{F}_{\mathsf{KES}}$

In Figure 7, we describe the ideal functionality $\mathcal{F}_{\mathsf{KES}}$ presented in [23], where $\mathcal{F}_{\mathsf{KES}}$ is used to sign the block.

Key evolving signature schemes formalize the notion of forward secure signature schemes. In forward secure signature schemes, compromise of the current secret key does not enable an adversary to forge signature pertaining to the past or rather the honest users can verify the a given signature was generated at a certain point in time, which can be guaranteed by evolving the signing key after each signature is generated and erasing the previous key in such a way that the actual signing key after for signing a message in the past cannot be recovered.

Definition 13 (Key Evolving Signature Schemes)**.** *A key evolving signature scheme is a quadruple of algorithms* KES = (Gen, Sign, Verify, Update)*, where:*

1) $\mathsf{Gen}(1^\lambda, T)$ *is a probabilistic algorithm which takes as input a security parameter* λ *and the total number of periods* T *and returns a pair* (sk_1, vk)*, the initial secret key and the public key;*

2) $\mathsf{Sign}(m)$ *takes as input the secret key* sk_j *for the time*

Functionality $\mathcal{F}_{\mathsf{DSIG}}$

$\mathcal{F}_{\mathsf{DSIG}}$ interacts with a signed U_S and stakeholder $U_1, ..., U_n$ as follows:

Key Generation. Upon receiving a message $(\mathsf{KeyGen}, sid, U_S)$ from a stakeholder U_S, verify that $sid = (U_S, sid')$ for some sid'. If not, then ignore the request. Else, hand $(\mathsf{KeyGen}, sid, U_S)$ to the adversary. Upon receiving $(\mathsf{PublicKey}, sid, U_S, v)$ from the adversary, send $(\mathsf{PublicKey}, sid, v)$ to U_S, record the entry (sid, U_S, v).

Signature Generation. Upon receiving a message $(\mathsf{Sign}, sid, U_S, m)$ from U_S, verify that (sid, U_S, v) is recorded for some sid. If not, then ignore the request. Else, send $(\mathsf{Sign}, sid, U_S, m)$ to the adversary. Upon receiving $(\mathsf{Signature}, sid, U_S, m, \sigma)$ from the adversary, verify that no entry $(m, \sigma, v, 0)$ is recorded. If it is, then output an error message to U_S and halt. Else, send $(\mathsf{Signature}, sid, m, \sigma)$ to U_S, and record the entry $(m, \sigma, v, 1)$.

Signature Verification. Upon receiving a message $(\mathsf{Verify}, sid, m, \sigma, v')$ from some stakeholder U_i, hand $(\mathsf{Verify}, sid, m, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, \phi)$ from the adversary do:

1) If $v' = v$ and the entry $(m, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the public key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2) Else, if $v' = v$, the signer is not corrupted, and no entry $(m, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, \sigma, v, 0)$. (This condition ensures unforgeability: If the public key v' is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
3) Else, if there is an entry (m, σ, v', f') recorded, then let $f = f'$. (This condition ensures consistency: All verification requests with identical parameters will result in the same answer.)
4) Else, Else, let $f = \phi$ and record the entry (m, σ, v', ϕ).

Output $(\mathsf{Verified}, sid, m, f)$ to U_i.

Figure 6: Functionality $\mathcal{F}_{\mathsf{DSIG}}$

period $j \leq T$ and a message m, outputting a signature σ_j on m for period j, and the period j is encoded in the signature itself.

3) *Verify(m, σ_j) is a deterministic verification algorithm that takes as input a public key vk, a message m and a signature σ_j, outputs 1 if σ_j is valid on m for time period j and 0 otherwise.*

4) *Update(sk_j) is a secret key update algorithm that takes as input a secret key sk_j for the current period j and outputs a new secret key sk_{j+1} for time period $j + 1$.*

The forward security of key evolving security is as follows:

Definition 14 (Forward Security)**.** *Formally, let the forger $F = (F_{cma}, F_{forge})$. F_{cma} has access to a signing oracle with adaptively chosen messages, and outputs (CM, b), where CM is the set of queried messages and b is the break-in time period. Given CM, $sign(CM)$ and the signing key sk_b for time period b, F_{forge} outputs $(m, \sigma_j) \leftarrow F_{forge}(CM, sign(CM), sk_b)$. F is successful if $(m, j) \notin CM$, $j < b$ and Verify$(m, \sigma_j) = 1$.*

A key evolving signature scheme KES is forward secure if the success probability of F is negligible in λ.

In [23], it is shown that a construction π_{KES} intuitively constructed from a key involving signature scheme such as [33][43] can realize $\mathcal{F}_{\mathsf{KES}}$.

D. Ideal Functionality $\mathcal{F}_{VRF}$

In Figure 8, we describe the ideal functionality $\mathcal{F}_{\mathsf{VRF}}$ presented in [23]. This functionality is used as a private test that is executed locally to decide whether a certain participant of the protocol is eligible to issue a block. $\mathcal{F}_{\mathsf{VRF}}$ is used to capture adaptive corruptions in [23][36] by guaranteeing that the adversary cannot predict the eligibility of a stakeholder to produce a block prior to corrupting it, thus he/she cannot gain an advantage by corrupting specific stakeholders.

Definition 15 (Verifiable Random Function)**.** *A function family $F_.(.)$:$\{0,1\}^l \rightarrow \{0,1\}^{\ell_{VRF}}$ is a family of VRFs if there exists algorithms (Gen,Prove,Ver) such that (i.) Gen(1^k) outputs a pair of keys ($VRF.pk$, $VRF.sk$), (ii.) Prove$_{VRF.sk}(x)$ outputs a pair ($F_{VRF.sk}(x)$,$\pi_{VRF.sk}(x)$), where $F_{VRF.sk}(x) \in \{0,1\}^{\ell_{VRF}}$ is the function value and $\pi_{VRF.sk}(x)$ is the proof of correctness, and (iii.) Ver$_{VRF.pk}(x, y, \pi_{VRF.sk}(x))$ verifies that $y = F_{VRF.sk}(x)$ using the proof $\pi_{VRF.sk}(x)$, outputting 1 if y is valid and 0 otherwise. Additionally, we require the following properties:*

1) **Uniqueness:** *no values ($VRF.pk$, x, y, y', $\pi_{VRF.sk}(x)$, $\pi_{VRF.sk}(x)'$) can satisfy Ver$_{VRF.pk}(x,y,\pi_{VRF.sk}(x))$=Ver$_{VRF.pk}(x,y',\pi_{VRF.sk}(x)')$ = 1 when $y \neq y'$.*
2) **Provability:** *if $(y, \pi_{VRF.sk}(x))$ = Prove$_{VRF.sk}(x)$, then we have Ver$_{VRF.pk}(x, y, \pi_{VRF.sk}(x)) = 1$.*
3) **Pseudorandomness:** *for any PPT adversary $\mathcal{A}$, set $y_0 = \{0,1\}^{\ell_{VRF}}$, $y_1 = F_{VRF.sk}(x)$ and $b \in \{0,1\}$, then provide y_b and the Prove oracle to $\mathcal{A}$, then $Pr[b = b' | b' \leftarrow \mathcal{A}(y_b, \mathsf{Prove})] \leq 1/2 + negl(\lambda)$.*

In addition, in [23], another property called **Unpredictability** *is also needed to guarantee by VRF to capture stronger attacks, namely if provided with an input that has high entropy, the output of the VRF is unpredictable even when the adversary is allowed to generate the secret key and public key pair. It was shown how to realize the $\mathcal{F}_{\mathsf{VRF}}$ in the random oracle based on the 2-Hash-DH verifiable oblivious PRF construction of [34] and we omit further details here.*

Functionality $\mathcal{F}_{\mathsf{KES}}$

$\mathcal{F}_{\mathsf{KES}}$ is parameterized by the total number of signature updates T, interacting with a signer U_S and stakeholder U_i as follows:

Key Generation. Upon receiving a message $(\mathsf{KeyGen}, sid, U_S)$ from a stakeholder U_S, verify that $sid = (U_S, sid')$ for some sid'. If not, then ignore the request. Else, send $(\mathsf{KeyGen}, sid, U_S)$ to the adversary. Upon receiving $(\mathsf{PublicKey}, sid, U_S, v)$ from the adversary, send $(\mathsf{PublicKey}, sid, v)$ to U_S, record the entry (sid, U_S, v) and set counter $\mathsf{k_{ctr}} = 1$.

Sign and Update. Upon receiving a message $(\mathsf{USign}, sid, U_S, m, j)$ from U_S, verify that (sid, U_S, v) is recorded for some sid and that $\mathsf{k_{ctr}} \leq j \leq T$. If not, then ignore the request. Else, set $\mathsf{k_{ctr}} = j + 1$ and send $(\mathsf{Sign}, sid, U_S, m, j)$ to the adversary. Upon receiving $(\mathsf{Signature}, sid, U_S, m, j, \sigma)$ from the adversary, verify that no entry $(m, j, \sigma, v, 0)$ is recorded. If it is, then output an error message to U_S and halt. Else, send $(\mathsf{Signature}, sid, m, j, \sigma)$ to U_S, record the entry $(m, j, \sigma, v, 1)$.

Signature Verification. Upon receiving a message $(\mathsf{Verify}, sid, m, j, \sigma, v')$ from some stakeholder U_i do:

1) If $v' = v$ and the entry $(m, j, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition ensures completeness: If the public key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
2) Else, if $v' = v$, the signer is not corrupted, and no entry $(m, j, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. (This condition ensures unforgeability: If the public key v' is the registered one, the signer is not corrupted, and m is never by signed by the signer, then the verification fails.)
3) Else, if there is an entry (m, j, σ, v', f') recorded, then let $f = f'$. (This condition ensures consistency: All verification requests with identical parameters will result in the same answer.)
4) Else, if $j < \mathsf{k_{ctr}}$, then let $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. Otherwise, if $j = \mathsf{k_{ctr}}$, send $(\mathsf{Verify}, sid, m, j, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, j, \phi)$ from the adversary, let $f = \phi$ and record the entry (m, j, σ, v', ϕ). (This condition ensures that the adversary is only able to forge signatures under keys belonging to corrupted parties for time periods corresponding to the current or future slots.)

Output $(\mathsf{Verified}, sid, m, j, f)$ to U_i.

Figure 7: Functionality $\mathcal{F}_{\mathsf{KES}}$

Functionality $\mathcal{F}_{\mathsf{VRF}}$

$\mathcal{F}_{\mathsf{VRF}}$ interacts with stakeholder $U_1, ..., U_n$ as follows:

Key Generation. Upon receiving a message (KeyGen, sid) from a stakeholder U_i, verify that $sid = (U_i, sid')$ for some sid'. If not, then ignore the request. Else, hand $(\mathsf{KeyGen}, sid, U_i)$ to the adversary. Upon receiving $(\mathsf{PublicKey}, sid, U_i, v)$ from the adversary, if U_i is honest, verify that v is unique, record the pair (U_i, v) and return $(\mathsf{PublicKey}, sid, v)$ to U_i. Initialize the table $T(v, .)$ to empty.

Malicious Key Generation. Upon receiving a message $(\mathsf{KeyGen}, sid, v)$ from $\mathcal{S}$, verify that v has not being recorded before; in this case initialize table $T(v, .)$ to empty and record the pair $(\mathcal{S}, v)$.

VRF Evaluation. Upon receiving a message (Eval, sid, m) from U_i, verify that some pair (U_i, v) is recorded. If not, then ignore the request. Then, if the value $T(v, m)$ is undefined, pick a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v, m) = (y, \emptyset)$. Then output $(\mathsf{Evaluated}, sid, y)$ to P, where y is such that $T(v, m) = (y, S)$ for some S.

VRF Evaluation and Proof. Upon receiving a message $(\mathsf{EvalProve}, sid, m)$ from U_i, verify that some pair (U_i, v) is recorded. If not, then ignore the request. Else, send $(\mathsf{EvalProve}, sid, U_i, m)$ to the adversary. Upon receiving $(\mathsf{Eval}, sid, m, \pi)$ from the adversary, if value $T(v, m)$ is undefined, verify that π is unique, pick a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v, m) = (y, \{\pi\})$. Else, if $T(v, m) = (y, S)$, set $T(v, m) = (y, S \cup \{\pi\})$. In any case, output $(\mathsf{Evaluated}, sid, y, \pi)$ to U_i.

Malicious VRF Evaluation. Upon receiving a message $(\mathsf{Eval}, sid, v, m)$ from $\mathcal{S}$ for some v, do the following. First, if $(\mathcal{S}, v)$ is recorded and $T(v, m)$ is undefined, then choose a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v, m) = (y, \emptyset)$. Then, if $T(v, m) = (y, S)$ for some $S \neq \emptyset$, output $(\mathsf{Evaluated}, sid, y)$ to $\mathcal{S}$, else ignore the request.

Verification. Upon receiving a message $(\mathsf{Verify}, sid, m, y, \pi, v')$ from some party P, send $(\mathsf{Verify}, sid, m, y, \pi, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, sid, m, y, \pi, v')$ from the adversary do:

1) If $v' = v$ for some (U_i, v) and the entry $T(U_i, m)$ equals (y, S) with $\pi \in S$, then set $f = 1$.
2) Else, if $v' = v$ for some (U_i, v), but no entry $T(U_i, m)$ of the form $(y, \{..., \pi, ...\})$ is recorded, then set $f = 0$.
3) Else, initialize the table $T(v', .)$ to empty, and set $f = 0$.

Output $(\mathsf{Verified}, sid, m, y, \pi, f)$ to P.

Figure 8: Functionality $\mathcal{F}_{\mathsf{VRF}}$